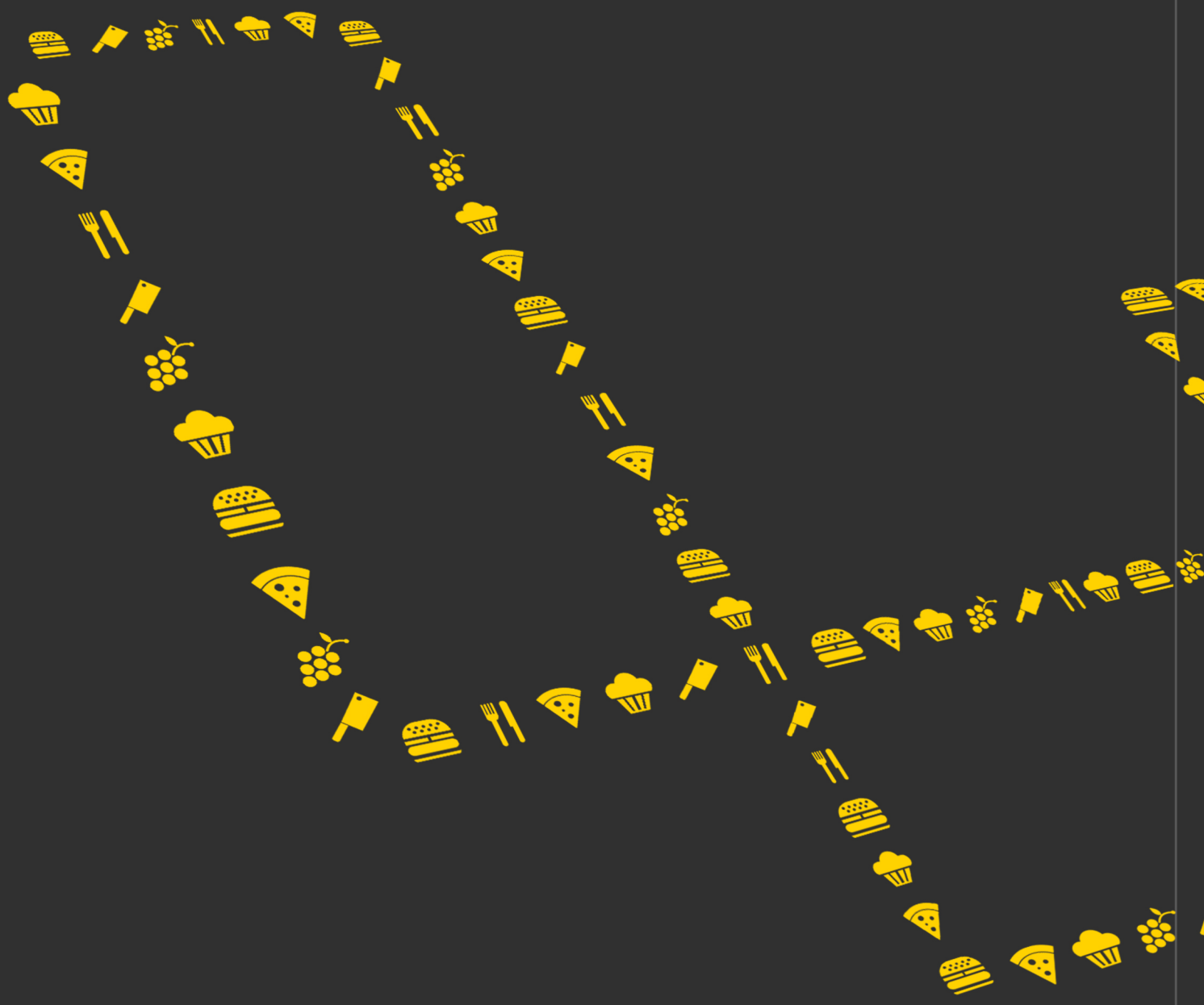

LARAVEL 4
COOKBOOK

BY CHRISTOPHER PITT

Laravel 4 Cookbook

Christopher Pitt and Taylor Otwell

This book is for sale at http://leanpub.com/laravel4cookbook

This version was published on 2013-11-20

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Christopher Pitt and Taylor Otwell by spreading the word about this book on Twitter!

The suggested hashtag for this book is #laravel4cookbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#laravel4cookbook

Contents

Dedication

I would like to thank Taylor Otwell for the Laravel framework. He may not have written part of this book (in the traditional sense), but without his tireless dedication to Laravel; none of this would have happened. I consider him a co-developer in the code I write daily.

I would like to thank all of my friends as Joe Public[n]. I have never worked anywhere I love more. You give me the freedom and encouragement to create, learn and have fun.

I would like to thank my family for their encouragement, support and general awesomeness.

I would like to thank my wife and children for being patient and loving me even when I'm not loveable.

I would like to thank Jesus. I encourage you to ask me why.

Forward

Hi, my name is Wayne Ashley Berry and I work with Chris at Joe Public where we write business critical software day in and day out. I've been writing software professionally for years… Chris is the guy I go to when Google doesn't have the answers.

What I love about Chris's work ethic is that he continually pushes the limits of software, frameworks and languages but then manages to hold back and use simple and understandable concepts.

B.B. King once said "Don't use the song to show off your skills, use your skills to show off the song.", Chris is like that Jazz musician who you know could out-play you with one hand but finds immense joy in playing four chord pop tracks.

Each case study in this book comes from hard earned experience. Consider each chapter years of experience, sleepless nights and stressful deadlines distilled into a set of best practices, common sense and good advice.

If you're looking to use Laravel, or even just PHP, for real-world projects then consider this book worth more that its file-size in gold.

What This Book Teaches

I'm writing this book (and the tutorials) in the hope that people can learn the things I have about Laravel 4. It's not meant as a replacement for any of the great Laravel books, but instead as a complement to the resources, documentation and framework.

This book teaches various aspects of Laravel 4 implementation, configuration and usage; as part of separate projects. The idea is not to demonstrate the only or best way to create any of these projects. It's not to show the only or best way to use Laravel 4. It's simply a different (and subjective) kind of documentation to the modularised version found at: **http://laravel.com/docs**

While this book touches on in the installation and hosting of Laravel applications; it's not an exhaustive reference for how to do these things. There are some instructions; which should be enough to get you up and running, but it assumes you are familiar with how things like LAMP (Linux, Apache, MySQL and PHP) work and are capable of installing and maintaining them.

Why Write This Book

I was learning how to use Laravel 4 more effectively, and found some subjects which I felt were worth sharing. I picked Medium (which later turned out to be a huge pain) and started putting a tutorial together. A few hours later I hit publish…

Then @laravelphp retweeted a link to the article. I think I spent the rest of the day just watching stats. The tutorial hit Medium's home page. It turns out there are a lot of people who wanted to know about Authentication (in Laravel), and just needed to be exposed to the article through @laravelphp's promotion of it.

Since then; I have been releasing a tutorial every two weeks.

The book grew out of the realisation that; while loads of people were reading the tutorials on Medium, some people weren't happy with the platform.

There are many compelling reasons for me to keep on using Medium to host the tutorials. I don't want to host my own thing because uptime is important, and outages in the night add years onto my life. The simple statistics and text formatting are also great.

I want to stay on Medium, but I also want people to want to read the tutorials and learn from them on other platforms. The book allows both of these things, as well as an important third thing…

The book is also intended as a means to give back to Laravel; in particular the invaluable work of Taylor Otwell. To this end, I have committed to give half of all sales to Taylor. The tutorials will always be free on Medium, and their content will mirror the chapters of this book (with obvious repetition omitted), but by purchasing this book you are helping to fund future Laravel development from him and tutorials from me.

Installing Laravel 4

Laravel 4 uses Composer to manage its dependencies. You can install Composer by following the instructions at **http://getcomposer.org/doc/00-intro.md#installation-nix**.

Once you have Composer working, make a new directory or navigation to an existing directory and install Laravel 4 with the following command:

```
composer create-project laravel/laravel ./ --prefer-dist
```

If you chose not to install Composer globally (though you really should), then the command you use should resemble the following:

```
php composer.phar create-project laravel/laravel ./ --prefer-dist
```

Both of these commands will start the process of installing Laravel 4. There are many dependencies to be sourced and downloaded; so this process may take some time to finish.

Authentication

If you're anything like me; you've spent a great deal of time building password-protected systems. I used to dread the point at which I had to bolt on the authentication system to a CMS or shopping cart. That was until I learned how easy it was with Laravel 4.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-authentication**

Configuring The Database

One of the best ways to manage users and authentication is by storing them in a database. The default Laravel 4 authentication mechanisms assume you will be using some form of database storage, and provides two drivers with which these database users can be retrieved and authenticated.

Connection To The Database

To use either of the provided drivers, we first need a valid connection to the database. Set it up by configuring and of the sections in the **app/config/database.php** file. Here's an example of the MySQL database I use for testing:

```
<?php

return [
    "fetch"       => PDO::FETCH_CLASS,
    "default"     => "mysql",
    "connections" => [
        "mysql" => [
                "driver"    => "mysql",
                "host"      => "localhost",
                "database"  => "tutorial",
                "username"  => "dev",
                "password"  => "dev",
                "charset"   => "utf8",
```

```
                "collation" => "utf8_unicode_ci",
                "prefix"    => ""
        ]
    ],
    "migrations" => "migration"
];
```

> This file should be saved as **app/config/database.php**.

> I have removed comments, extraneous lines and superfluous driver configuration options.

Database Driver

The first driver which Laravel 4 provides is a called **database**. As the name suggests; this driver queries the database directly, in order to determine whether users matching provided credentials exist, and whether the appropriate authentication credentials have been provided.

If this is the driver you want to use; you will need the following database table in the database you have already configured:

```
CREATE TABLE `user` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `username` varchar(255) DEFAULT NULL,
    `password` varchar(255) DEFAULT NULL,
    `email` varchar(255) DEFAULT NULL,
    `created_at` datetime DEFAULT NULL,
    `updated_at` datetime DEFAULT NULL,
    PRIMARY KEY (`id`)
) CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

> Here, and further on, I deviate from the standard of plural database table names. Usually, I would recommend sticking with the standard, but this gave me an opportunity to demonstrate how you

> can configure database table names in both migrations and models.

Eloquent Driver

The second driver which Laravel 4 provides is called eloquent. Eloquent is the name of the ORM which Laravel 4 also provides, for abstracting model data. It is similar in that it will ultimately query a database to determine whether a user is authentic, but the interface which it uses to make that determination is quite different from direct database queries.

If you're building medium-to-large applications, using Laravel 4, then you stand a good chance of using Eloquent models to represent database objects. It is with this in mind that I will spend some time elaborating on the involvement of Eloquent models in the authentication process.

> If you want to ignore all things Eloquent; feel free to skip the following sections dealing with migrations and models.

Creating A Migration

Since we're using Eloquent to manage how our application communicates with the database; we may as well use Laravel 4's database table manipulation tools.

To get started, navigate to the root of your project and type the following command:

```
1 php artisan migrate:make --table="user" CreateUserTable
```

> The **–table="user"** flag matches the **$table=user** property we will define in the **User** model.

This will generate the scaffolding for the users table, which should resemble the following:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUserTable
extends Migration
{
    public function up()
    {
        Schema::table('user', function(Blueprint $table)
        {
            //
        });
    }
    public function down()
    {
        Schema::table('user', function(Blueprint $table)
        {
            //
        });
    }
}
```

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateUserTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

The file naming scheme may seem odd, but it is for a good reason. Migration systems are designed to be able to run on any server, and the order in which they must run is fixed. All of this is to allow changes to the database to be version-controlled.

The migration is created with just the most basic scaffolding, which means we need to add the fields for the users table:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUserTable
extends Migration
{
    public function up()
    {
        Schema::create("user", function(Blueprint $table)
        {
            $table->increments("id");

            $table
                ->string("username")
                ->nullable()
                ->default(null);

            $table
                ->string("password")
                ->nullable()
                ->default(null);

            $table
                ->string("email")
                ->nullable()
                ->default(null);

            $table
                ->dateTime("created_at")
                ->nullable()
                ->default(null);

            $table
                ->dateTime("updated_at")
                ->nullable()
                ->default(null);
        });
    }

    public function down()
```

```
43      {
44          Schema::dropIfExists("user");
45      }
46  }
```

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateUserTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

Here; we've added fields for id, username, password, date created and date updated. There are methods to shortcut the timestamp fields, but I prefer to add these fields explicitly. All the fields are nullable and their default value is null.

We've also added the drop method, which will be run if the migrations are reversed; which will drop the users table if it exists.

> The shortcut for adding the timestamp fields can be found at: **http://laravel.com/docs/schema#adding-columns**

This migration will work, even if you only want to use the database driver, but it's usually part of a larger setup; including models and seeders.

Creating A Model

Laravel 4 provides a **User** model, with all the interface methods it requires. I have modified it slightly, but the basics are still there...

```
1   <?php
2
3   use Illuminate\Auth\UserInterface;
4   use Illuminate\Auth\Reminders\RemindableInterface;
5
6   class User
7   extends Eloquent
8   implements UserInterface, RemindableInterface
9   {
10      protected $table  = "user";
```

```
11      protected $hidden = ["password"];
12
13      public function getAuthIdentifier()
14      {
15          return $this->getKey();
16      }
17
18      public function getAuthPassword()
19      {
20          return $this->password;
21      }
22
23      public function getReminderEmail()
24      {
25          return $this->email;
26      }
27  }
```

> This file should be saved as **app/models/User.php**.

> Note the **$table=user** property we have defined. It should match the table we defined in our migrations.

The **User** model extends **Eloquent** and implements two interfaces which ensure the model is valid for authentication and reminder operations. We'll look at the interfaces later, but its important to note the methods these interfaces require.

Laravel 4 allows the user of either email address or username with which to identify the user, but it is a different field from that which the **getAuthIdentifier()** returns. The **UserInterface** interface does specify the password field name, but this can be changed by overriding/changing the **getAuthPassword()** method.

The **getReminderEmail()** method returns an email address with which to contact the user with a password reset email, should this be required.

You are otherwise free to specify any model customisation, without fear it will break the built-in authentication mechanisms.

Creating A Seeder

Laravel 4 also includes seeding system, which can be used to add records to your database after initial migration. To add the initial users to my project, I have the following seeder class:

```
<?php

class UserSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $users = [
            [
                "username" => "christopher.pitt",
                "password" => Hash::make("7h3iMOST!53cu23"),
                "email"    => "chris@example.com"
            ]
        ];

        foreach ($users as $user)
        {
            User::create($user);
        }
    }
}
```

> This file should be saved as **app/database/seeds/UserSeeder.php**.

Running this will add my user account to the database, but in order to run this; we need to add it to the main **DatabaseSeeder** class:

```php
<?php

class DatabaseSeeder
extends Seeder
{
    public function run()
    {
        Eloquent::unguard();
        $this->call("UserSeeder");
    }
}
```

> This file should be saved as **app/database/seeds/DatabaseSeeder.php**.

Now, when the **DatabaseSeeder** class is invoked; it will seed the users table with my account. If you've already set up your migration and model, and provided valid database connection details, then the following commands should get everything up and running.

```
composer dump-autoload
php artisan migrate
php artisan db:seed
```

The first command makes sure all the new classes we've created are correctly autoloaded. The second creates the database tables specified for the migration. The third seeds the user data into the users table.

Configuring Authentication

The configuration options for the authentication mechanisms are sparse, but they do allow for some customisation.

```php
<?php

return [
    "driver"   => "eloquent",
    "model"    => "User",
    "reminder" => [
        "email"  => "email.request",
        "table"  => "token",
        "expire" => 60
    ]
];
```

> This file should be saved as **app/config/auth.php**.

All of these settings are important, and most are self-explanatory. The view used to compose the request email is specified by **email ⇒ email.request** and the time in which the reset token will expire is specified by **expire ⇒ 60**.

> Pay particular attention to the view specified by **email ⇒ email.request**—it tells Laravel to load the file **app/views/email/request.blade.php** instead of the default **app/views/emails/auth/reminder.blade.php**.

> There are various things that would benefit from configuration options; which are currently being hard-coded in the providers. We will look at some of these, as they come up.

Logging In

To allow authentic users to use our application, we're going to build a login page; where users can enter their login details. If their details are valid, they will be redirected to their profile page.

Creating A Layout View

Before we create any of the pages for out application; it would be wise to abstract away all of our layout markup and styling. To this end; we will create a layout view with various includes, using the Blade tempting engine.

First off, we need to create the layout view.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <link
            type="text/css"
            rel="stylesheet"
            href="/css/layout.css" />
        <title>
            Tutorial
        </title>
    </head>
    <body>
        @include("header")
        <div class="content">
            <div class="container">
                @yield("content")
            </div>
        </div>
        @include("footer")
    </body>
</html>
```

> This file should be saved as **app/views/layout.blade.php**.

The layout view is mostly standard HTML, with two Blade-specific tags in it. The **@include()** tags tell Laravel to include the views (named in those strings; as **header** and **footer**) from the views directory.

Notice how we've omitted the **.blade.php** extension? Laravel automatically adds this on for us. It also binds the data provided to the layout view to both includes.

The second Blade tag is **yield()**. This tag accepts a section name, and outputs the data stored in that section. The views in our application will extend this layout view; while specifying their own **content** sections so that their markup is embedded in the markup of the layout. You'll see exactly how sections are defined shortly.

```
@section("header")
    <div class="header">
        <div class="container">
            <h1>Tutorial</h1>
        </div>
    </div>
@show
```

> This file should be saved as **app/views/header.blade.php**.

The header include file contains two blade tags which, together, instruct Blade to store the markup in the named section, and render it in the template.

```
@section("footer")
    <div class="footer">
        <div class="container">
            Powered by <a href="http://laravel.com/">Laravel</a>
        </div>
    </div>
@show
```

> This file should be saved as **app/views/footer.blade.php**.

Similarly, the footer include wraps its markup in a named section and immediately renders it in the template.

You may be wondering why we would need to wrap the markup, in these include files, in sections. We are rendering them immediately, after all. Doing this allows us to alter their contents. We will see this in action soon.

```
body
{
    margin      : 0;
    padding     : 0 0 50px 0;
    font-family : "Helvetica", "Arial";
    font-size   : 14px;
    line-height : 18px;
    cursor      : default;
}
a
{
    color : #ef7c61;
}
.container
{
    width    : 960px;
    position : relative;
    margin   : 0 auto;
}
.header, .footer
{
    background  : #000;
    line-height : 50px;
    height      : 50px;
    width       : 100%;
    color       : #fff;
}
.header h1, .header a
{
    display : inline-block;
}
.header h1
{
    margin      : 0;
    font-weight : normal;
}
.footer
{
    position : absolute;
    bottom   : 0;
}
.content
```

```
{
    padding : 25px 0;
}
label, input, .error
{
    clear   : both;
    float   : left;
    margin : 5px 0;
}
.error
{
    color : #ef7c61;
}
```

This file should be saved as **public/css/layout.css**.

We finish by adding some basic styles; which we linked to in the **head** element. These alter the default fonts and layout. Your application would still work without them, but it would just look a little messy.

Creating A Login View

The login view is essentially a form; in which users enter their credentials.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/login",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("username", "Username") }}
        {{ Form::text("username", Input::old("username"), [
            "placeholder" => "john.smith"
        ]) }}
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "□□□□□□□□□□□"
        ]) }}
```

```
15          {{ Form::submit("login") }}
16      {{ Form::close() }}
17  @stop
18  @section("footer")
19      @parent
20      <script src="//polyfill.io"></script>
21  @stop
```

> This file should be saved as **app/views/user/login.blade.php**.

The first Blade tag, in the login view, tells Laravel that this view extends the layout view. The second tells it what markup to include in the content section. These tags will form the basis for all the views (other than layout) we will be creating.

We then use **{{** and **}}** to tell Laravel we want the contained code to be interpreted as PHP. We open the form with the **Form::open()** method; providing a route for the form to post to, and optional parameters in the second argument.

We then define two labels and three inputs. The labels accept a name argument, followed by a text argument. The text input accepts a name argument, a default value argument and and optional parameters. The password input accepts a name argument and optional parameters. Lastly, the submit input accepts a name argument and a text argument (similar to labels).

We close out the form with a call to **Form::close()**.

> You can find out more about the **Form** methods Laravel offers at: **http://laravel.com/docs/html**

The last part of the login view is where we override the default footer markup (specified in the footer include we created earlier). We use the same section name, but we don't end the section with **@show**. It will already render due to how we defined the include, so we just use **@stop** in the same way as we closed the content section.

We also use the **@parent** Blade tag to tell Laravel we want the markup we defined in the default footer to display. We're not completely changing it, only adding a script tag.

> You can find out more about Blade tags at: **http://laravel.com/docs/templates#blade-templating**

The script we included is called polyfill.io. It's a collection of browser shims which allow things like the **placeholder** attribute (which aren't always present in older browsers).

> You can find out more about Polyfill.io at: **https://github.com/jonathantneal/polyfill**

Our login view is now complete, but basically useless without the server-side code to accept the input and return a result. Let's get that sorted!

Creating A Login Action

The login action is what glues the authentication logic to the views we have created. If you have been following along, you might have wondered when we were going to try any of this stuff out in a browser. Up to this point; there was nothing telling our application to load that view.

To begin with; we need to add a route for the login action.

```
1  <?php
2
3  Route::any("/", [
4      "as"   => "user/login",
5      "uses" => "UserController@loginAction"
6  ]);
```

> This file should be saved as **app/routes.php**.

The routes file displays a holding page for a new Laravel 4 application, by rendering a view directly. We need to change that to use a controller/action. It's not that we have to—we could just as easily perform the logic in the routes file—it just wouldn't be very tidy.

We specify a name for the route with **as** ⇒ **user/login**, and give it a destination with **uses** ⇒ **UserController@loginAction**. This will match all calls to the default route /, and even has a name which we can use to refer back to this route easily.

Next up, we need to create the controller.

```php
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        return View::make("user/login");
    }
}
```

> This file should be saved as **app/controllers/UserController.php**.

We define the **UserController** (to extend the **Controller** class). In it; we have the single **loginAction()** method we specified in the routes file. All this currently does is render the login view to the browser, but it's enough for us to be able to see our progress!

Authenticating Users

Right, so we've got the form and now we need to tie it into the database so we can authenticate users correctly.

```php
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
```

```
17                  echo "Validation passed!";
18              }
19              else
20              {
21                  echo "Validation failed!";
22              }
23          }
24
25          return View::make("user/login");
26      }
27  }
```

> This file should be saved as **app/controllers/UserController.php**.

Our **UserController** class has changed somewhat. Firstly, we need to act on data that is posted to the **loginAction()** method; and to do that we check the server property **REQUEST_METHOD**. If this value is **POST** we can assume that the form has been posted to this action, and we proceed to the validation phase.

> It's also common to see separate get and post actions for the same page. While this makes things a little neater, and avoids the need for checking the **REQUEST_METHOD** property; I prefer to handle both in the same action.

Laravel 4 provides a great validation system, and one of the ways to use it is by calling the **Validator::make()** method. The first argument is an array of data to validate, and the second argument is an array of rules.

We have only specified that the username and password fields are required, but there are many other validation rules (some of which we will use in a while). The **Validator** class also has a **passes()** method, which we use to tell whether the posted form data is valid.

> Sometimes it's better to store the validation logic outside of the controller. I often put it in a model, but you could also create a class specifically for handling and validating input.

If you post this form; it will now tell you whether the required fields were supplied or not, but there is a more elegant way to display this kind of message...

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $data = [];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                //
            }
            else
            {
                $data["errors"] = new MessageBag([
                    "password" => [
                        "Username and/or password invalid."
                    ]
                ]);
            }
        }

        return View::make("user/login", $data);
    }
}
```

> This file should be saved as **app/controllers/UserController.php**.

With the changes above; we're using the **MessageBag** class to store validation error messages. This is similar to how the Validation class implicitly stores its errors, but instead of showing individual error messages for either username or password; we're showing a single error message for both. Login forms are a little more secure that way!

To display this error message, we also need to change the login view.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/login",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("username", "Username") }}
        {{ Form::text("username", Input::get("username"), [
            "placeholder" => "john.smith"
        ]) }}
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "••••••••••"
        ]) }}
        @if ($error = $errors->first("password"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::submit("login") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

> This file should be saved as **app/views/user/login.blade.php**.

As you can probably see; we've added a check for the existence of the error message, and rendered it within a styled div element. If validation fails, you will now see the error message below the password field.

Redirecting With Input

One of the common pitfalls of forms is how refreshing the page most often re-submits the form. We can overcome this with some Laravel magic. We'll store the posted form data in the session, and redirect back to the login page!

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

```

```
28              if ($validator->passes())
29              {
30                  //
31              }
32              else
33              {
34                  $data["errors"] = new MessageBag([
35                      "password" => [
36                          "Username and/or password invalid."
37                      ]
38                  ]);
39
40                  $data["username"] = Input::get("username");
41
42                  return Redirect::route("user/login")
43                      ->withInput($data);
44              }
45          }
46
47          return View::make("user/login", $data);
48      }
49  }
```

This file should be saved as **app/controllers/UserController.php**.

The first thing we've done is to declare a new **MessageBag** instance. We do this because the view will still check for the errors **MessageBag**, whether or not it has been saved to the session. If it is, however, in the session; we overwrite the new instance we created with the stored instance.

We then add it to the **$data** array so that it is passed to the view, and can be rendered.

If the validation fails; we save the username to the **$data** array, along with the validation errors, and we redirect back to the same route (also using the **withInput()** method to store our data to the session).

Our view remains unchanged, but we can refresh without the horrible form re-submission (and the pesky browser messages that go with it).

Authenticating Credentials

The last step in authentication is to check the provided form data against the database. Laravel handles this easily for us.

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                $credentials = [
                    "username" => Input::get("username"),
                    "password" => Input::get("password")
                ];

                if (Auth::attempt($credentials))
                {
                    return Redirect::route("user/profile");
                }
```

```
39              }
40
41              $data["errors"] = new MessageBag([
42                  "password" => [
43                      "Username and/or password invalid."
44                  ]
45              ]);
46
47              $data["username"] = Input::get("username");
48
49              return Redirect::route("user/login")
50                  ->withInput($data);
51          }
52
53          return View::make("user/login", $data);
54      }
55  }
```

This file should be saved as **app/controllers/UserController.php**.

We simply need to pass the posted form data (**$credentials**) to the **Auth::attempt()** method and, if the user credentials are valid, the user will be logged in. If valid, we return a redirect to the user profile page.

We have also removed the errors code outside of the else clause. This is so that it will occur both on validation errors as well as authentication errors. The same error message (in the case of login pages) is fine.

Resetting Passwords

The password reset mechanism built into Laravel 4 is great! We're going to set it up so users can reset their passwords just by providing their email address.

Creating A Password Reset View

We need two views for users to be able to reset their passwords. We need a view for them to enter their email address so they can be sent a reset token, and we need a view for them to enter a new password for their account.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/request",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

This file should be saved as **app/views/user/request.blade.php**.

This view is similar to the login view, except it has a single field for an email address.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::route("user/reset") . $token,
        "autocomplete" => "off"
    ]) }}
        @if ($error = $errors->first("token"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        @if ($error = $errors->first("email"))
            <div class="error">
```

```
                {{ $error }}
            </div>
        @endif
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "••••••••••"
        ]) }}
        @if ($error = $errors->first("password"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("password_confirmation", "Confirm") }}
        {{ Form::password("password_confirmation", [
            "placeholder" => "••••••••••"
        ]) }}
        @if ($error = $errors->first("password_confirmation"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

> This file should be saved as **app/views/user/reset.blade.php**.

Ok, you get it by now. There's a form with some inputs and error messages. One important thing to note is the change in form action; namely the use of **URL::route()** in combination with a variable assigned to the view. We will set that in the action, so don't worry about it for now.

I've also slightly modified the password token request email, though it remains mostly the same as the default view provided by new Laravel 4 installations.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
    </head>
    <body>
        <h1>Password Reset</h1>
        To reset your password, complete this form:
        {{ URL::route("user/reset") . "?token=" . $token }}
    </body>
</html>
```

> This file should be saved as **app/views/email/request.blade.php**.

> Remember we changed the configuration options for emailing this view from the default **app/views/emails/auth/reminder.blade.php**.

Creating A Password Reset Action

In order for the actions to be accessible; we need to add routes for them.

```
<?php

Route::any("/", [
    "as"   => "user/login",
    "uses" => "UserController@loginAction"
]);

Route::any("/request", [
    "as"   => "user/request",
    "uses" => "UserController@requestAction"
]);

Route::any("/reset", [
```

```
    "as"   => "user/reset",
    "uses" => "UserController@resetAction"
]);
```

> This file should be saved as **app/routes.php**.

Remember; the request route is for requesting a reset token, and the reset route is for resetting a password.

We also need to generate the password reset tokens table; using artisan.

```
php artisan auth:reminders
```

This will generate a migration template for the reminder table.

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTokenTable
extends Migration
{
    public function up()
    {
        Schema::create("token", function(Blueprint $table)
        {
            $table
                ->string("email")
                ->nullable()
                ->default(null);

            $table
                ->string("token")
                ->nullable()
                ->default(null);

            $table
                ->timestamp("created_at")
```

```
25                  ->nullable()
26                  ->default(null);
27          });
28      }
29
30      public function down()
31      {
32          Schema::dropIfExists("token");
33      }
34  }
```

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateTokenTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

I've modified the template slightly, but the basics are all the same. This will create a table with **email**, **token** and **created_at** fields; which the authentication mechanisms use to generate and validate password reset tokens.

With these in place, we can begin to add our password reset actions.

```
1   public function requestAction()
2   {
3       $data = [
4           "requested" => Input::old("requested")
5       ];
6
7       if (Input::server("REQUEST_METHOD") == "POST")
8       {
9           $validator = Validator::make(Input::all(), [
10              "email" => "required"
11          ]);
12
13          if ($validator->passes())
14          {
15              $credentials = [
16                  "email" => Input::get("email")
17              ];
18
19              Password::remind($credentials,
```

```
                function($message, $user)
                {
                    $message->from("chris@example.com");
                }
            );

            $data["requested"] = true;

            return Redirect::route("user/request")
                ->withInput($data);
        }
    }

    return View::make("user/request", $data);
}
```

This was extracted from **app/controllers/UserController.php**.

The **requestAction()** method validates the posted form data in much the same was as the **loginAction()** method, but instead of passing the form data to **Auth::attempt()**, it passes it to **Password::remind()**. This method accepts an array of credentials (which usually just includes an email address), and also allows an optional callback in which you can customise the email that gets sent out.

```
public function resetAction()
{
    $token = "?token=" . Input::get("token");

    $errors = new MessageBag();

    if ($old = Input::old("errors"))
    {
        $errors = $old;
    }

    $data = [
        "token"  => $token,
        "errors" => $errors
    ];
```

```
16
17      if (Input::server("REQUEST_METHOD") == "POST")
18      {
19          $validator = Validator::make(Input::all(), [
20              "email"                 => "required|email",
21              "password"              => "required|min:6",
22              "password_confirmation" => "same:password",
23              "token"                 => "exists:token,token"
24          ]);
25
26          if ($validator->passes())
27          {
28              $credentials = [
29                  "email" => Input::get("email")
30              ];
31
32              Password::reset($credentials,
33                  function($user, $password)
34                  {
35                      $user->password = Hash::make($password);
36                      $user->save();
37
38                      Auth::login($user);
39                      return Redirect::route("user/profile");
40                  }
41              );
42          }
43
44          $data["email"] = Input::get("email");
45
46          $data["errors"] = $validator->errors();
47
48          return Redirect::to(URL::route("user/reset") . $token)
49              ->withInput($data);
50      }
51
52      return View::make("user/reset", $data);
53  }
```

> This was extracted from **app/controllers/UserController.php**.

The **resetAction()** method is much the same. We begin it by creating the token query string (which we use for redirects, to maintain the token in all states of the reset page). We fetch old error messages, as we did for the login page, and we validate the posted form data.

If all the data is valid, we pass it to **Password::reset()**. The second argument is the logic used to update the user's database record. We're updating the password, saving the record and then automatically logging the user in.

If all of that went down without a hitch; we redirect to the profile page. If not; we redirect back to the reset page, passing along the error messages.

> There is one strange thing about the authentication mechanisms here; the password/token field names are hard-coded and there is hard-coded validation built into the **Password::reset()** function which does not use the **Validation** class. So long as your field names are **password**, **password_confirmation** and **token**, and your password is longer than 6 characters, you shouldn't notice this strange thing.
>
> Alternatively, you can modify field names and validation applied in the **vendor/laravel/framework/src/Illuminate/Auth/Reminders/PasswordBroker.php** file or implement your own **ReminderServiceProvider** to replace that which Laravel 4 provides. The details for both of those approaches are beyond the scope of this tutorial. You can find details for creating service providers in Taylor Otwell's excellent book, at: **https://leanpub.com/laravel**

> As I mentioned before, you can set the amount of time after which the password reset tokens expire; in the **app/config/auth.php** file.

> You can find out more about the authentication methods at: **http://laravel.com/docs/security#authenticating-users**

> You can find out more about the mail methods at: **http://laravel.com/docs/mail**

Working With Authenticated Users

Ok. We've got login and password reset under our belt. The final part of this tutorial is for us to use the user session data in our application, and protect unauthenticated access to secure parts of our application.

Creating A Profile Page

To show off some of the user session data we have access to; we've going to implement the profile view.

```
@extends("layout")
@section("content")
    <h2>Hello {{ Auth::user()->username }}</h2>
    <p>Welcome to your sparse profile page.</p>
@stop
```

> This file should be saved as **app/views/user/profile.blade.php**.

This incredibly sparse profile page shows off a single thing; you can get data from the user model by accessing object returned by the **Auth::user()** method. Any fields you have defined on this model (or database table) are accessible in this way.

```
public function profileAction()
{
    return View::make("user/profile");
}
```

> This was extracted from **app/controllers/UserController.php**.

The **profileAction()** method is just a simple as the view. We don't need to pass any data to the view, or even get hold of the user session using any special code. **Auth::user()** does it all!

In order for this page to be accessible, we do need to add a route for it. We're going to do that in a minute; but now would be a good time to talk about protecting sensitive pages of our application...

Creating Filters

Laravel 4 includes a filters file, in which we can define filters to run for single (or even groups of) routes.

```
<?php

Route::filter("auth", function()
{
    if (Auth::guest())
    {
        return Redirect::route("user/login");
    }
});

Route::filter("guest", function()
{
    if (Auth::check())
    {
        return Redirect::route("user/profile");
    }
});

Route::filter("csrf", function()
{
    if (Session::token() != Input::get("_token"))
    {
        throw new Illuminate\Session\TokenMismatchException;
    }
});
```

> This file should be saved as **app/filters.php**.

The first filter is for routes (or pages if you prefer) for which a user must be authenticated. The second is for the exact opposite; for which users must not be authenticated. The last filter is one that we have been using all along.

When we use the **Form::open()** method; Laravel automatically adds a hidden field to our forms. This field contains a special security token which is checked each time a form is submitted. You don't really need to understand why this is more secure...

> ...but if you want to, read this: **http://blog.ircmaxell.com/2013/02/preventing-csrf-attacks.html**

In order to apply these filters, we need to modify our routes file.

```
1  <?php
2
3  Route::group(["before" => "guest"], function()
4  {
5      Route::any("/", [
6          "as"   => "user/login",
7          "uses" => "UserController@loginAction"
8      ]);
9
10     Route::any("/request", [
11         "as"   => "user/request",
12         "uses" => "UserController@requestAction"
13     ]);
14
15     Route::any("/reset", [
16         "as"   => "user/reset",
17         "uses" => "UserController@resetAction"
18     ]);
19 });
20
21 Route::group(["before" => "auth"], function()
22 {
23     Route::any("/profile", [
```

```
        "as"   => "user/profile",
        "uses" => "UserController@profileAction"
    ]);

    Route::any("/logout", [
        "as"   => "user/logout",
        "uses" => "UserController@logoutAction"
    ]);
});
```

This file should be saved as **app/routes.php**.

To protect parts of our application, we group groups together with the **Route::group()** method. The first argument lets us specify which filters to apply to the enclosed routes. We want to group all of our routes in which users should not be authenticated; so that users don't see them when logged in. We do the opposite for the profile page because only authenticated users should get to see their profile pages.

Creating A Logout Action

To test these new security measures out (and to round off the tutorial) we need to create a **logoutAction()** method and add links to the header so that users can log out.

```
public function logoutAction()
{
    Auth::logout();
    return Redirect::route("user/login");
}
```

This was extracted from **app/controllers/UserController.php**.

The **logoutAction()** method calls the **Auth::logout()** method to close the user session, and redirects back to the login screen. Easy as pie!

This is what the new header include looks like:

```
@section("header")
    <div class="header">
        <div class="container">
            <h1>Tutorial</h1>
            @if (Auth::check())
                <a href="{{ URL::route("user/logout") }}">
                    logout
                </a>
                |
                <a href="{{ URL::route("user/profile") }}">
                    profile
                </a>
            @else
                <a href="{{ URL::route("user/login") }}">
                    login
                </a>
            @endif
        </div>
    </div>
@show
```

This file should be saved as **app/views/header.blade.php**.

Access Control List

Previously we looked at how to set up a basic authentication system. In this chapter; we're going to continue to improve the authentication system by adding what's called ACL (Access Control List) to the authentication layer.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-acl**

Managing Groups

We're going to be creating an interface for adding, modifying and deleting user groups. Groups will be the containers to which we add various users and resources. We'll do that by create a migration and a model for groups, but we're also going to optimise the way we create migrations.

Refactoring Migrations

We've got a few more migrations to create in this tutorial; so it's a good time for us to refactor our approach to creating them...

```
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
4  use Illuminate\Database\Schema\Blueprint;
5
6  class BaseMigration
7  extends Migration
8  {
9      protected $table;
10
11     public function getTable()
12     {
13         if ($this->table == null)
14         {
15             throw new Exception("Table not set.");
```

```
        }

        return $this->table;
    }

    public function setTable(Blueprint $table)
    {
        $this->table = $table;
        return $this;
    }

    public function addNullable($type, $key)
    {
        $types = [
            "boolean",
            "dateTime",
            "integer",
            "string",
            "text"
        ];

        if (in_array($type, $types))
        {
            $this->getTable()
                ->{$type}($key)
                ->nullable()
                ->default(null);
        }

        return $this;
    }

    public function addTimestamps()
    {
        $this->addNullable("dateTime", "created_at");
        $this->addNullable("dateTime", "updated_at");
        $this->addNullable("dateTime", "deleted_at");
        return $this;
    }

    public function addPrimary()
    {
```

```
58          $this->getTable()->increments("id");
59          return $this;
60      }
61
62      public function addForeign($key)
63      {
64          $this->addNullable("integer", $key);
65          $this->getTable()->index($key);
66          return $this;
67      }
68
69      public function addBoolean($key)
70      {
71          return $this->addNullable("boolean", $key);
72      }
73
74      public function addDateTime($key)
75      {
76          return $this->addNullable("dateTime", $key);
77      }
78
79      public function addInteger($key)
80      {
81          return $this->addNullable("integer", $key);
82      }
83
84      public function addString($key)
85      {
86          return $this->addNullable("string", $key);
87      }
88
89      public function addText($key)
90      {
91          return $this->addNullable("text", $key);
92      }
93  }
```

This file should be saved as **app/database/migrations/BaseMigration.php**.

We're going to base all of our models off of a single **BaseModel** class. This will make it possible for us to reuse a lot of the repeated code we had before.

The **BaseModel** class has a single protected **$table** property, for storing the current **Blueprint** instance we are giving inside our migration callbacks. We have a typical setter for this; and an atypical getter (which throws an exception if **$this->table** hasn't been set). We do this as we need a way to validate that the methods which require a valid **Blueprint** instance have one or throw an exception.

Our **BaseMigration** class also has a factory method for creating fields of various types. If the type provided is one of those defined; a nullable field of that type will be created. This significantly shortens the code we used previously to create nullable fields.

Following this; we have **addPrimary()**, **addForeign()** and **addTimestamps()**. The **addPrimary()** method is a bit clearer than the **increments()** method, the **addForeign()** method adds both a nullable integer field and an index for the foreign key. The **addTimestamps()** method is similar to the **Blueprint**'s **timestamps()** method; except that it also adds the **deleted_at** timestamp field.

Finally; there are a handful of methods which proxy to the **addNullable()** method.

Using these methods, the amount of code required for the migrations we will create (and have already created) is drastically reduced.

```
1  <?php
2
3  use Illuminate\Database\Schema\Blueprint;
4
5  class CreateGroupTable
6  extends BaseMigration
7  {
8      public function up()
9      {
10         Schema::create("group", function(Blueprint $table)
11         {
12             $this
13                 ->setTable($table)
14                 ->addPrimary()
15                 ->addString("name")
16                 ->addTimestamps();
17         });
18     }
19
20     public function down()
21     {
22         Schema::dropIfExists("group");
```

```
    }
}
```

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateGroupTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

The group table has a primary key, timestamp fields (including **created_at**, **updated_at** and **deleted_at**) as well as a name field.

If you're skipping migrations; the following SQL should create the same table structure as the migration:

```
CREATE TABLE `group` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `name` varchar(255) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB CHARSET=utf8;
```

Listing Groups

We're going to be creating views to manage group records; more comprehensive than those we created for users previously, but much the same in terms of complexity.

```
@extends("layout")
@section("content")
    @if (count($groups))
        <table>
            <tr>
                <th>name</th>
            </tr>
            @foreach ($groups as $group)
                <tr>
                    <td>{{ $group->name }}</td>
                </tr>
            @endforeach
        </table>
```

```
    @else
        <p>There are no groups.</p>
    @endif
    <a href="{{ URL::route("group/add") }}">add group</a>
@stop
```

This file should be saved as **app/views/group/index.blade.php**.

The first view is the index view. This should list all the groups that are in the database. We extend the layout as usual, defining a content block for the markup specific to this page.

The main idea is to iterate over the group records, but before we do that we first check if there are any groups. After all, we don't want to go to the trouble of showing a table if there's nothing to put in it.

If there are groups, we create a (rough) table and iterate over the groups; creating a row for each. We finish off the view by adding a link to create a new group.

```
Route::any("/group/index", [
    "as"   => "group/index",
    "uses" => "GroupController@indexAction"
]);
```

This was extracted from **app/routes.php**.

```
<?php

class Group
extends Eloquent
{
    protected $table = "group";

    protected $softDelete = true;

    protected $guarded = [
```

```
        "id",
        "created_at",
        "updated_at",
        "deleted_at"
    ];
}
```

This file should be saved as **app/models/Group.php**.

```
<?php

class GroupController
extends Controller
{
   public function indexAction()
   {
        return View::make("group/index", [
            "groups" => Group::all()
        ]);
    }
}
```

This file should be saved as **app/controllers/GroupController.php**.

In order to view the index page; we need to define a route to it. We also need to define a model for the group table. Lastly; we render the index view, having passed all the groups to the view. Navigating to this route should now display the message "There are no groups." as we have yet to add any.

You can test out how the index page looks by adding a group to the database directly.

An important thing to note is the use of **$softDelete**. Laravel 4 provides a new method of ensuring that no data is hastily deleted via Eloquent; so long as this property is set. If **true**; any calls to

the **$group->delete()** method will set the **deleted_at** timestamp to the date and time on which the method was invoked. Records with a **deleted_at** timestamp (which is not **null**) will not be returned in normal **QueryBuilder** (including Eloquent) queries.

> You can find out more about soft deleting at: **http://laravel.com/docs/eloquent#soft-deleting**

Another important thing to note is the use of **$guarded**. Laravel 4 provides mass assignment protection. What we're doing by specifying this list of fields; is telling Eloquent which fields should not be settable when providing an array of data in the creation of a new **Group** instance.

> You can find out more about this mass assignment protection at: **http://laravel.com/docs/eloquent#mass-assignment**

Adding Groups

We're going to be abstracting much of the validation out of the controllers and into new form classes.

```
1  <?php
2
3  use Illuminate\Support\MessageBag;
4
5  class BaseForm
6  {
7      protected $passes;
8      protected $errors;
9
10     public function __construct()
11     {
12         $errors = new MessageBag();
13
14         if ($old = Input::old("errors"))
15         {
16             $errors = $old;
17         }
18
```

```
        $this->errors = $errors;
    }

    public function isValid($rules)
    {
        $validator = Validator::make(Input::all(), $rules);
        $this->passes = $validator->passes();
        $this->errors = $validator->errors();
        return $this->passes;
    }

    public function getErrors()
    {
        return $this->errors;
    }

    public function setErrors(MessageBag $errors)
    {
        $this->errors = $errors;
        return $this;
    }

    public function hasErrors()
    {
        return $this->errors->any();
    }

    public function getError($key)
    {
        return $this->getErrors()->first($key);
    }

    public function isPosted()
    {
        return Input::server("REQUEST_METHOD") == "POST";
    }
}
```

This file should be saved as **app/forms/BaseForm.php**.

The **BaseForm** class checks for the error messages we would normally store to flash (session) storage. We would typically pull this data in each action, and now it will happen when each form class instance is created.

The validation takes place in the **isValid()** method, which gets all the input data and compares it to a set of provided validation rules. This will be used later, in **BaseForm** subclasses.

BaseForm also has a few methods for managing the **$errors** property, which should always be a **MessageBag** instance. They can be used to set and get the **MessageBag** instance, get an individual message and even tell whether there are any error messages present.

There's also a method to determine whether the request method, for the current request, is POST.

```
<?php

class GroupForm
extends BaseForm
{
    public function isValidForAdd()
    {
        return $this->isValid([
            "name" => "required"
        ]);
    }

    public function isValidForEdit()
    {
        return $this->isValid([
            "id"   => "exists:group,id",
            "name" => "required"
        ]);
    }

    public function isValidForDelete()
    {
        return $this->isValid([
            "id" => "exists:group,id"
        ]);
    }
}
```

> This file should be saved as **app/forms/GroupForm.php**.

The first implementation of **BaseForm** is the **GroupForm** class. It's quite simply by comparison; defining three validation methods. These will be used in their respective actions.

We also need a way to generate not only validation error message markup but also a quicker way to create form markup. Laravel 4 has great utilities for creating form and HTML markup, so let's see how these can be extended.

```
{{ Form::label("name", "Name") }}
{{ Form::text("name", Input::old("name"), [
    "placeholder" => "new group"
]) }}
```

We've already seen this type of Blade template syntax before. The label and text helpers are great for programatically creating the markup we would otherwise have to create; but sometimes it is nice to be able to create our own markup generators for commonly repeated patterns.

What if we, for instance, often use a combination of label, text and error message markup? It would then be ideal for us to create what's called a macro to generate that markup.

```
<?php

Form::macro("field", function($options)
{
    $markup = "";

    $type = "text";

    if (!empty($options["type"]))
    {
        $type = $options["type"];
    }

    if (empty($options["name"]))
    {
        return;
    }

    $name = $options["name"];
```

```php

    $label = "";

    if (!empty($options["label"]))
    {
        $label = $options["label"];
    }

    $value = Input::old($name);

    if (!empty($options["value"]))
    {
        $value = Input::old($name, $options["value"]);
    }

    $placeholder = "";

    if (!empty($options["placeholder"]))
    {
        $placeholder = $options["placeholder"];
    }

    $class = "";

    if (!empty($options["class"]))
    {
        $class = " " . $options["class"];
    }

    $parameters = [
        "class"       => "form-control" . $class,
        "placeholder" => $placeholder
    ];

    $error = "";

    if (!empty($options["form"]))
    {
        $error = $options["form"]->getError($name);
    }

    if ($type !== "hidden")
```

```
    {
        $markup .= "<div class='form-group";
        $markup .= ($error ? " has-error" : "");
        $markup .= "'>";
    }

    switch ($type)
    {
        case "text":
        {
            $markup .= Form::label($name, $label, [
                "class" => "control-label"
            ]);

           $markup .= Form::text($name, $value, $parameters);

           break;
        }

        case "password":
        {
            $markup .= Form::label($name, $label, [
                "class" => "control-label"
            ]);

            $markup .= Form::password($name, $parameters);

            break;
        }

        case "checkbox":
        {
            $markup .= "<div class='checkbox'>";
            $markup .= "<label>";
            $markup .= Form::checkbox($name, 1, !!$value);
            $markup .= " " . $label;
            $markup .= "</label>";
            $markup .= "</div>";

            break;
        }

```

```
104         case "hidden":
105         {
106             $markup .= Form::hidden($name, $value);
107             break;
108         }
109     }
110 
111     if ($error)
112     {
113         $markup .= "<span class='help-block'>";
114         $markup .= $error;
115         $markup .= "</span>";
116     }
117 
118     if ($type !== "hidden")
119     {
120         $markup .= "</div>";
121     }
122 
123     return $markup;
124 });
```

> This file should be saved as **app/macros.php**.

This macro evaluates an **$options** array, generating a label, input element and validation error message. There's white a lot of checking involved to ensure that all the required data is there, and that optional data affects the generated markup correctly. It supports text inputs, password inputs, checkboxes and hidden fields; but more types can easily be added.

> The markup this macro generates is Bootstrap friendly. If you haven't already heard of Bootstrap (where have you been?) then you can find out more about it at: **http://getbootstrap.com/**

To see this in action, we need to include it in the startup processes of the application and then modify the form views to use it:

```
require app_path() . "/macros.php";
```

> This was extracted from **app/start/global.php**.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "group/add",
        "autocomplete" => "off"
    ]) }}
        {{ Form::field([
            "name"        => "name",
            "label"       => "Name",
            "form"        => $form,
            "placeholder" => "new group"
        ])}}
        {{ Form::submit("save") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

> This file should be saved as **app/views/group/add.blade.php**.

You'll notice how much neater the view is; thanks to the form class handling the error messages for us. This view happens to be relatively short since there's only a single field (name) for groups.

```
.help-block
{
    float : left;
    clear : left;
}

.form-group.has-error .help-block
{
    color : #ef7c61;
}
```

> This was extracted from **public/css/layout.css**.

One last thing we have to do, to get the error messages to look the same as they did before, is to add a bit of CSS to target the Bootstrap-friendly error messages.

With the add view complete; we can create the **addAction()** method:

```
public function addAction()
{
    $form = new GroupForm();

    if ($form->isPosted())
    {
        if ($form->isValidForAdd())
        {
            Group::create([
                "name" => Input::get("name")
            ]);

            return Redirect::route("group/index");
        }

        return Redirect::route("group/add")->withInput([
            "name"   => Input::get("name"),
            "errors" => $form->getErrors()
        ]);
    }

```

```
22      return View::make("group/add", [
23          "form" => $form
24      ]);
25  }
```

> This was extracted from **app/controllers/GroupController.php**.

You can also see how much simpler our **addAction()** method is; now that we're using the **GroupForm** class. It takes care of retrieving old error messages and handling validation so that we can simply create groups and redirect.

Editing Groups

The view and action for editing groups is much the same as for adding groups.

```
1   @extends("layout")
2   @section("content")
3       {{ Form::open([
4           "url"          => URL::full(),
5           "autocomplete" => "off"
6       ]) }}
7           {{ Form::field([
8               "name"        => "name",
9               "label"       => "Name",
10              "form"        => $form,
11              "placeholder" => "new group",
12              "value"       => $group->name
13          ]) }}
14          {{ Form::submit("save") }}
15      {{ Form::close() }}
16  @stop
17  @section("footer")
18      @parent
19      <script src="//polyfill.io"></script>
20  @stop
```

This file should be saved as **app/views/group/edit.blade.php**.

The only difference here is the form action we're setting. We need to take into account that a group id will be provided to the edit page, so the URL must be adjusted to maintain this id even after the form is posted. For that; we use the **URL::full()** method which returns the full, current URL.

```
1  public function editAction()
2  {
3      $form = new GroupForm();
4
5      $group = Group::findOrFail(Input::get("id"));
6      $url   = URL::full();
7
8      if ($form->isPosted())
9      {
10         if ($form->isValidForEdit())
11         {
12             $group->name = Input::get("name");
13             $group->save();
14             return Redirect::route("group/index");
15         }
16
17         return Redirect::to($url)->withInput([
18             "name"   => Input::get("name"),
19             "errors" => $form->getErrors(),
20             "url"    => $url
21         ]);
22     }
23
24     return View::make("group/edit", [
25         "form"  => $form,
26         "group" => $group
27     ]);
28 }
```

This was extracted from **app/controllers/GroupController.php**.

In the **editAction()** method; we're still create a new instance of **GroupForm**. Because we're editing a group, we need to get that group to display its data in the view. We do this with Eloquent's **findOrFail()** method; which will cause a 404 error page to be displayed if the id is not found within the database.

The rest of the action is much the same as the **addAction()** method. We'll also need to add the edit route to the **routes.php** file...

```
Route::any("/group/edit", [
    "as"   => "group/edit",
    "uses" => "GroupController@editAction"
]);
```

> This was extracted from **app/routes.php**.

Deleting Groups

There are a number of options we can explore when creating the delete interface, but we'll go with the quickest which is just to present a link on the listing page.

```
@extends("layout")
@section("content")
    @if (count($groups))
        <table>
            <tr>
                <th>name</th>
                <th> </th>
            </tr>
            @foreach ($groups as $group)
                <tr>
                    <td>{{ $group->name }}</td>
                    <td>
                        <a href="{{ URL::route("group/edit") }}?id={{ $group->id \
}}">edit</a>
                        <a href="{{ URL::route("group/delete") }}?id={{ $group->i\
d }}" class="confirm" data-confirm="Are you sure you want to delete this group?">\
delete</a>
                    </td>
```

```
                </tr>
            @endforeach
        </table>
    @else
        <p>There are no groups.</p>
    @endif
    <a href="{{ URL::route("group/add") }}">add group</a>
@stop
```

This file should be saved as **app/views/group/index.blade.php**.

We've modified the **group/index** view to include two links; which will redirect users either to the edit page or the delete action. Notice the **class="confirm"** and **data-confirm="..."** attributes we've added to the delete link—we'll use these shortly. We'll also need to add the delete route to the **routes.php** file...

```
Route::any("/group/delete", [
    "as"   => "group/delete",
    "uses" => "GroupController@deleteAction"
]);
```

This was extracted from **app/routes.php**.

Since we've chosen such an easy method of deleting groups, the action is pretty straightforward:

```
public function deleteAction()
{
    $form = new GroupForm();

    if ($form->isValidForDelete())
    {
        $group = Group::findOrFail(Input::get("id"));
        $group->delete();
    }

```

```
    return Redirect::route("group/index");
}
```

> This was extracted from **app/controllers/GroupController.php**.

We simply need to find a group with the provided id (using the **findOrFail()** method we saw earlier) and delete it. After that; we redirect back to the listing page. Before we take this for a spin, let's add the following JavaScript:

```
(function($){
    $(".confirm").on("click", function() {
        return confirm($(this).data("confirm"));
    });
}(jQuery));
```

> This file should be saved as **public/js/layout.js**.

```
@section("footer")
    @parent
    <script src="/js/jquery.js"></script>
    <script src="/js/layout.js"></script>
@stop
```

> This was extracted from **app/views/group/index.blade.php**.

You'll notice I have linked to **jquery.js** (any recent version will do). The code in **layout.js** adds a click event handler on to every element with **class="confirm"** to prompt the user with the message in **data-confirm="..."**. If "OK" is clicked; the callback returns **true** and the browser will redirect to the page on the other end (in this case the **deleteAction()** method on our **GroupController** class). Otherwise the click will be ignored.

Adding Users And Resources

Next on our list is making a way for us to specify resource information and add users to our groups. Both of these thing will happen on the group edit page; but before we get there we will need to deal with migrations, models and relationships...

Adding Migrations, Models And Relationships

```
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateResourceTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("resource", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addString("name")
                ->addString("pattern")
                ->addString("target")
                ->addBoolean("secure")
                ->addTimestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists("resource");
    }
}
```

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateResourceTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

> We are calling them resources to avoid the name collision with the existing Route class.

If you're skipping migrations; the following SQL should create the same table structure as the migration:

```
CREATE TABLE `resource` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `name` varchar(255) DEFAULT NULL,
 `pattern` varchar(255) DEFAULT NULL,
 `target` varchar(255) DEFAULT NULL,
 `secure` tinyint(1) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB CHARSET=utf8;
```

The resource table has fields for the things we usually store in our routes file. The idea is that we keep the route information in the database so we can both programatically generate the routes for our application; and so that we can link various routes to groups for controlling access to various parts of our application.

```
<?php

class Resource
extends Eloquent
{
    protected $table = "resource";

    protected $softDelete = true;

    protected $guarded = [
        "id",
        "created_at",
        "updated_at",
        "deleted_at"
    ];

    public function groups()
```

```
18      {
19          return $this->belongsToMany("Group")->withTimestamps();
20      }
21  }
```

> This file should be saved as **app/models/Resource.php**.

The Resource model is similar to those we've seen before; but it also specifies a many-to-many relationship (in the **groups()** method). This will allows us to return related groups with **$this->groups**. We'll use that later!

> The **withTimestamps()** method will tell Eloquent to update the timestamps of related groups when resources are updated. You can find out more about it at: **http://laravel.com/docs/eloquent#working-with-pivot-tables**

We also need to add the reverse relationship to the **Group** model:

```
1  public function resources()
2  {
3      return $this->belongsToMany("Resource")->withTimestamps();
4  }
```

> This was extracted from **app/models/Group.php**.

> There really is a lot to relationships in Eloquent; more than we have time to cover now. I will be going into more detail about these relationships in future tutorials; exploring the different types and configuration options. For now, this is all we need to complete this chapter.

We can also define relationships for users and groups, as in the following examples:

```
public function users()
{
    return $this->belongsToMany("User")->withTimestamps();
}
```

> This was extracted from **app/models/Group.php**.

```
public function groups()
{
    return $this->belongsToMany("Group")->withTimestamps();
}
```

> This was extracted from **app/models/User.php**.

Before we're quite done with the database work; we'll also need to remember to set up the pivot tables in which the relationship data will be stored.

```
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateGroupUserTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("group_user", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addForeign("group_id")
                ->addForeign("user_id")
                ->addTimestamps();
```

```
        });
    }

    public function down()
    {
        Schema::dropIfExists("group_user");
    }
}
```

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateGroupUserTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

```
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateGroupResourceTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("group_resource", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addForeign("group_id")
                ->addForeign("resource_id")
                ->addTimestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists("group_resource");
    }
}
```

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateGroupResourceTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

We now have a way to manage the data relating to groups; so let's create the views and actions through which we can capture this data.

If you're skipping migrations; the following SQL should create the same table structures as the migrations:

```
CREATE TABLE `group_resource` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `group_id` int(11) DEFAULT NULL,
 `resource_id` int(11) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`),
 KEY `group_resource_group_id_index` (`group_id`),
 KEY `group_resource_resource_id_index` (`resource_id`)
) ENGINE=InnoDB CHARSET=utf8;

CREATE TABLE `group_user` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `group_id` int(11) DEFAULT NULL,
 `user_id` int(11) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`),
 KEY `group_user_group_id_index` (`group_id`),
 KEY `group_user_user_id_index` (`user_id`)
) ENGINE=InnoDB CHARSET=utf8;
```

Adding Views

The views we need to create are those in which we will select which users and resources should be assigned to a group.

```
<div class="assign">
    @foreach ($resources as $resource)
        <div class="checkbox">
            {{ Form::checkbox("resource_id[]", $resource->id, $group->resources->\
contains($resource->id)) }}
            {{ $resource->name }}
        </div>
    @endforeach
</div>
```

This file should be saved as **app/views/resource/assign.blade.php**.

```
<div class="assign">
    @foreach ($users as $user)
        <div class="checkbox">
            {{ Form::checkbox("user_id[]", $user->id, $group->users->contains($us\
er->id)) }}
            {{ $user->username }}
        </div>
    @endforeach
</div>
```

This file should be saved as **app/views/user/assign.blade.php**.

These views similarly iterate over resources and users (passed to the group edit view) and render markup for checkboxes. It's important to note the names of the checkbox inputs ending in **[]**—this is the recommended way to passing array-like data in HTML forms.

The first parameter of the **Form::checkbox()** method is the input's name. The second is its value. The third is whether of not the checkbox should initially be checked. Eloquent models provide a useful **contains()** method which searches the related rows for those matching the provided id(s).

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::full(),
        "autocomplete" => "off"
    ]) }}
        {{ Form::field([
            "name"        => "name",
            "label"       => "Name",
            "form"        => $form,
            "placeholder" => "new group",
            "value"       => $group->name
        ])}}
        @include("user/assign")
        @include("resource/assign")
        {{ Form::submit("save") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

> This file should be saved as **app/views/group/edit.blade.php**.

We've modified the **group/edit** view to include the new assign views. If you try to edit a group, at this point, you might see an error. This is because we still need to pass the users and resources to the view...

```
return View::make("group/edit", [
    "form"      => $form,
    "group"     => $group,
    "users"     => User::all(),
    "resources" => Resource::where("secure", true)->get()
]);
```

This was extracted from **app/controllers/GroupController.php**.

Seeding Resources

We return all the users (so that any user can be in any group) and the resources that need to be secure. Right now, that database table is empty, but we can easily create a seeder for it:

```
<?php

class ResourceSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $resources = [
            [
                "pattern" => "/",
                "name"    => "user/login",
                "target"  => "UserController@loginAction",
                "secure"  => false
            ],
            [
                "pattern" => "/request",
                "name"    => "user/request",
                "target"  => "UserController@requestAction",
                "secure"  => false
            ],
            [
                "pattern" => "/reset",
                "name"    => "user/reset",
                "target"  => "UserController@resetAction",
                "secure"  => false
            ],
            [
                "pattern" => "/logout",
                "name"    => "user/logout",
                "target"  => "UserController@logoutAction",
                "secure"  => true
```

```
            ],
            [
                "pattern" => "/profile",
                "name"    => "user/profile",
                "target"  => "UserController@profileAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/index",
                "name"    => "group/index",
                "target"  => "GroupController@indexAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/add",
                "name"    => "group/add",
                "target"  => "GroupController@addAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/edit",
                "name"    => "group/edit",
                "target"  => "GroupController@editAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/delete",
                "name"    => "group/delete",
                "target"  => "GroupController@deleteAction",
                "secure"  => true
            ]
        ];

        foreach ($resources as $resource)
        {
            Resource::create($resource);
        }
    }
}
```

> This file should be saved as **app/database/seeds/ResourceSeeder.php**.

We should also add this seeder to the **DatabaseSeeder** class so that the Artisan commands which deal with seeding pick it up:

```
<?php

class DatabaseSeeder
extends Seeder
{
    public function run()
    {
        Eloquent::unguard();

        $this->call("ResourceSeeder");
        $this->call("UserSeeder");
    }
}
```

> This file should be saved as **app/database/seeds/DatabaseSeeder.php**.

Saving Relationships

Now you should be seeing the lists of resources and users when you try to edit a group. We need to save selections when the group is saved; so that we can successfully assign both users and resources to groups.

```php
if ($form->isValidForEdit())
{
    $group->name = Input::get("name");
    $group->save();

    $group->users()->sync(Input::get("user_id", []));
    $group->resources()->sync(Input::get("resource_id", []));

    return Redirect::route("group/index");
}
```

> This was extracted from **app/controllers/GroupController.php**.

Laravel 4 provides and excellent method for synchronising related database records—the **sync()** method. You simply provide it with the id(s) of the related records and it makes sure there is a record for each relationship. It couldn't be easier!

Finally, we will add a bit of CSS to make the lists less of a mess...

```css
.assign
{
    padding     : 10px 0 0 0;
    line-height : 22px;
}
.checkbox, .assign
{
    float : left;
    clear : left;
}
.checkbox input[type='checkbox']
{
    margin : 0 10px 0 0;
    float  : none;
}
```

> This was extracted from **public/css/layout.css**.

Take it for a spin! You will find that the related records are created (in the pivot) tables, and each time you submit it; the edit page will remember the correct relationships and show them back to you.

Advanced Routes

The final thing we need to do is manage how resources are translated into routes and how the security behaves in the presence of our simple ACL.

```
<?php

Route::group(["before" => "guest"], function()
{
    $resources = Resource::where("secure", false)->get();

    foreach ($resources as $resource)
    {
        Route::any($resource->pattern, [
            "as"   => $resource->name,
            "uses" => $resource->target
        ]);
    }
});

Route::group(["before" => "auth"], function()
{
    $resources = Resource::where("secure", true)->get();

    foreach ($resources as $resource)
    {
        Route::any($resource->pattern, [
            "as"   => $resource->name,
            "uses" => $resource->target
        ]);
    }
});
```

> This file should be saved as **app/routes.php**.

There are some significant changes to the routes file. Firstly, all the routes are being generated from resources. We no longer need to hard-code routes in this file because we can save them in the database.

> It's more efficient hard-coding them, and we really should be caching them if we have to read them from the database; but that's the subject of future tutorials—we've running out of time here!

All the "insecure" routes are rendered in the first block—the block in which routes are subject to the **guest** filter. All the "secure" routes are rendered in the secure; where they are subject to the **auth** filter.

```
Route::filter("auth", function()
{
    if (Auth::guest())
    {
        return Redirect::route("user/login");
    }
    else
    {
        foreach (Auth::user()->groups as $group)
        {
            foreach ($group->resources as $resource)
            {
                $path = Route::getCurrentRoute()->getPath();

                if ($resource->pattern == $path)
                {
                    return;
                }
            }
        }

        return Redirect::route("user/login");
    }
});
```

> This was extracted from **app/filters.php**.

The new **auth** filter needs not only to make sure the user is authenticated, but also that one of the group to which they are assigned has the current route assigned to it also. Users can belong to multiple groups and so can resources; so this is the only (albeit inefficient way) to filter allowed resources from those which the user is not allowed access to.

To test this out; alter the group to which your user account belongs to disallow access to the **group/add** route. When you try to visit it you will be redirected first to the **user/login** route and the not the **user/profile** route.

> You need to make sure you're not disallowing access to a route specified in the auth filter. That will probably lead to a redirect loop!

Lastly, we need a way to hide links to disallowed resources...

```
<?php

if (!function_exists("allowed"))
{
    function allowed($route)
    {
        if (Auth::check())
        {
            foreach (Auth::user()->groups as $group)
            {
                foreach ($group->resources as $resource)
                {
                    if ($resource->name == $route)
                    {
                        return true;
                    }
                }
            }
        }

        return false;
```

```
22      }
23  }
```

> This file should be saved as **app/helpers.php**.

```
1  require app_path() . "/helpers.php";
```

> This was extracted from **app/start/global.php**.

Once we've included that **helpers.php** file in the startup processes of our application; we can check whether the authenticated user is allowed access to resources simply by passing the resource name to the **allowed()** method.

> The first time you run the migrations (if you're installing from GitHub); you may see errors relating to the resource table. This is likely caused by the routes.php file trying to load routes from an empty database table before seeding takes place. Try commenting out the code in routes.php until you've successfully migrated and seeded your database. There are nicer ways to do this; all of which I will not cover now.

Try this out by wrapping the links of your application in a condition which references this method.

> For the sake of brevity; I have not included any examples of this, though many can be found, in the source code, on GitHub.

Deployment

There are few things which have improved my life quite as much as learning how to create a custom deployment process for my projects. Nothing is worse than having to worry about how to get your files onto a remote server, when you've got an important bug to fix.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-deployment**

Deployment processes are one of the most subjective things about development. Everyone's got their own ideas about what should and shouldn't be done. There are sometimes best practises; though these tend only to apply to subsets of the whole process.

Things get even more tricky when it comes to deploying Laravel 4 applications because there aren't really any best practises to speak of, when it comes to working with remote servers and deploying code.

Remember this as you continue—this tutorial isn't the only approach to deploying Laravel 4 applications. It's simply a process I've found works for me.

Dependencies

Our deployment processes will need to handle JavaScript and CSS files. Assetic is an asset management library which we will use to do all the heavy lifting in this area.

To install it; we need to add two requirements to our composer.json file:

```
1  "kriswallsmith/assetic"   : "1.2.*@dev",
2  "toopay/assetic-minifier" : "dev-master"
```

> This was extracted from **composer.json**.

Lastly, we will also be using Jason Lewis' Resource Watcher library, which integrates with Laravel 4's **Filesystem** classes to enable notification of file changes. You'll see why that's useful in a bit...

```
"jasonlewis/resource-watcher" : "dev-master"
```

> This was extracted from **composer.json**.

Once these requirements are added to the **composer.json** file, we need to update the vendor folder:

```
composer update
```

We'll look at how to integrate these into our deployment workflow shortly.

Environment Commands

Environments are a small, yet powerful, aspect of any Laravel 4 application. They primarily allow the specification of machine-based configuration options.

There are a few things you need to know about environments, in Laravel 4:

1. The files contained in the root of **app/config** are merged or overridden by environment-based configuration files.
2. Configuration files that are specific to an environment are stored in folders matching their environment name.
3. Environments are determined by an array specified in **bootstrap/start.php** and are matched according to the name of the machine on which the application is being run.
4. An application can have any number of environments; each with their own configuration files. There can also be multiple machine names (hosts) in each environment. You can have two staging servers, a production server and a testing server (for instance). If their machine names match those in **bootstrap/start.php** then their individual configuration files will be loaded.
5. All Artisan commands can be given an **–env** option which will override the environment settings of the machine on which the commands are run. I'm sure there are other ways in which environments affect application execution, but you get the point: environments are a big-little thing.

Checking Environments

As I mentioned earlier; environments are usually specified in **boostrap/start.php**. This is probably going to be ok for the 99% of Laravel 4 applications that will ever be made, but we can improve upon it slightly still.

We're going to make the list of environments somewhat dynamic, and load them in a slightly different way to how they are loaded out-the-box.

The first thing we're going to do is learn how to make a command to tell us what the current environment is. Commands are the Laravel 4 way of extending the power and functionality of the Artisan command line tool. There are some commands already available when installing Laravel 4 (and we've seen some of them already, in previous tutorials).

To make our own, we can use an Artisan command:

```
1  php artisan command:make FooCommand
```

This command will make a new file at **app/commands/FooCommand .php**. Inside this file you will begin to see what commands look like under the hood. Here's an example of the file that gets generated:

```
1  <?php
2
3  use Illuminate\Console\Command;
4  use Symfony\Component\Console\Input\InputOption;
5  use Symfony\Component\Console\Input\InputArgument;
6
7  class FooCommand extends Command {
8
9      /**
10     * The console command name.
11     *
12     * @var string
13     */
14     protected $name = 'command:name';
15
16     /**
17     * The console command description.
18     *
19     * @var string
20     */
21     protected $description = 'Command description.';
22
```

```
23      /**
24      * Create a new command instance.
25      *
26      * @return void
27      */
28      public function __construct()
29      {
30          parent::__construct();
31      }
32
33      /**
34      * Execute the console command.
35      *
36      * @return void
37      */
38      public function fire()
39      {
40          //
41      }
42
43      /**
44      * Get the console command arguments.
45      *
46      * @return array
47      */
48      protected function getArguments()
49      {
50          return array(
51              array(
52                  'example',
53                  InputArgument::REQUIRED,
54                  'An example argument.'
55              ),
56          );
57      }
58
59      /**
60      * Get the console command options.
61      *
62      * @return array
63      */
64      protected function getOptions()
```

```
65      {
66          return array(
67              array(
68                  'example',
69                  null,
70                  InputOption::VALUE_OPTIONAL,
71                  'An example option.',
72                  null
73              ),
74          );
75      }
76
77  }
```

> This file should be saved as **app/commands/FooCommand.php**.

There are a few things of importance here:

1. The **$name** property is used both to describe the command as well as invoke it. If we change it to **foo**, and register it correctly (as we'll do in a moment), then we would be able to call it with: **php artisan foo**
2. The description property is only descriptive. When all the registered command are displayed (by a call to: **php artisan**) then this description will be shown next to the name of the command.
3. The **fire()** method is where all the action happens. If you want your command to do anything; there is where it needs to get done.
4. The **getArguments()** method should return an array of arguments (or parameters) to the command. If our command was: **php artisan foo bar**, then **bar** would be an argument. Arguments are named (which we will see shortly).
5. The **getOptions()** method should return an array of options (or flags) to the command. If out command was: **php artisan foo –baz**, then **–baz** would be an option. Options are also named (which we will also see shortly). This file gives us a good starting point from which to build our own set of commands.

We begin our commands by creating the **EnvironmentCommand** class:

```php
<?php

use Illuminate\Console\Command;

class EnvironmentCommand
extends Command
{
    protected $name = "environment";

    protected $description = "Lists environment commands.";

    public function fire()
    {
        $this->line(trim("
            <comment>environment:get</comment>
            <info>gets host and environment.</info>
        "));

        $this->line(trim("
            <comment>environment:set</comment>
            <info>adds host to environment.</info>
        "));

        $this->line(trim("
            <comment>environment:remove</comment>
            <info>removes host from environment.</info>
        "));
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [];
    }
}
```

> This file should be saved as **app/commands/EnvironmentCommand.php**.

The command class begins with us setting a useful name and description for the following commands we will create. The **fire()** method includes three calls to the **line()** method; which prints text to the command line. The **getArguments()** and **getOptions()** methods return empty arrays because we do not expect to handle any arguments or options.

> You may have noticed the XML notation within the calls to the line() method. Laravel 4's console library extends Symphony 2's console library; and Symphony 2's console library allows these definitions in order to alter the meaning and appearance of text rendered to the console.
>
> While the appearance will be changed, by using this notation, it's not the best thing to be doing (semantically speaking). The bits in the comment elements aren't comments any more than the bit in the info elements are.
>
> We're simply using it for a refreshing variation in the console text colours. If this sort of jacky behaviour offends your senses, feel free to omit the XML notation altogether!

Before we can run any commands; we need to register them with Artisan:

```
1  Artisan::add(new EnvironmentCommand);
```

> This was extracted from **app/start/artisan.php**.

There's nothing much more to say than; this code registers the command with Artisan, so that it can be invoked. We should be able to call the command now, and it should show us something like the following:

```
1  □ php artisan environment
2  environment:get gets host and environment.
3  environment:set adds host to environment.
4  environment:remove removes host from environment.
```

Congratulations! We've successfully created and registered our first Artisan command. Let's go ahead and make a few more.

To make the first described command (**environment:get**); we're going to subclass the **EnvironmentCommand** class we just created. We'll be adding reusable code in the **EnvironmentCommand** class so it's a means of accessing this code in related commands.

```
<?php

use Illuminate\Console\Command;

class EnvironmentGetCommand
extends EnvironmentCommand
{
    protected $name = "environment:get";

    protected $description = "Gets host and environment.";

    public function fire()
    {
        $this->line(trim("
            <comment>Host:</comment>
            <info>" . $this->getHost() . "</info>
        "));

        $this->line(trim("
            <comment>Environment:</comment>
            <info>" . $this->getEnvironment() . "</info>
        "));
    }
}
```

This file should be saved as **app/commands/EnvironmentGetCommand.php**.

The **EnvironmentGetCommand** does slightly more than the previous command we made. It fetches the host name and the environment name from functions we must define in the **EnvironmentCommand** class:

```
protected function getHost()
{
    return gethostname();
}

protected function getEnvironment()
{
    return App::environment();
}
```

> This was extracted from **app/commands/EnvironmentCommand.php**.

The **gethostname()** function returns the name of the machine on which it is invoked. Similarly; the **App::environment()** method returns the name of the environment in which Laravel is being run.

After registering and running this command, I see the following:

```
 php artisan environment:get
Host: formativ.local
Environment: local
```

Setting Environments

The next command is going to allow us to alter these values from the command line (without changing hard-coded values)...

```
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class EnvironmentSetCommand
extends EnvironmentCommand
{
    protected $name = "environment:set";

    protected $description = "Adds host to environment.";

```

```
14      public function fire()
15      {
16          $host        = $this->getHost();
17          $config      = $this->getConfig();
18          $overwrite   = $this->option("host");
19          $environment = $this->argument("environment");
20
21          if (!isset($config[$environment]))
22          {
23              $config[$environment] = [];
24          }
25
26          $use = $host;
27
28          if ($overwrite)
29          {
30              $use = $overwrite;
31          }
32
33          if (!in_array($use, $config[$environment]))
34          {
35              $config[$environment][] = $use;
36          }
37
38          $this->setConfig($config);
39
40          $this->line(trim("
41              <info>Added</info>
42              <comment>" . $use . "</comment>
43              <info>to</info>
44              <comment>" . $environment . "</comment>
45              <info>environment.</info>
46          "));
47      }
48
49      protected function getArguments()
50      {
51          return [
52              [
53                  "environment",
54                  InputArgument::REQUIRED,
55                  "Environment to add the host to."
```

```
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "host",
                null,
                InputOption::VALUE_OPTIONAL,
                "Host to add.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/EnvironmentSetCommand.php**.

The **EnvironmentSetCommand** class' **fire()** method begins by getting the host name and a configuration array (using inherited methods). It also checks for a host option and an environment argument.

If the host option is provided; it will be added to the list of hosts for the provided environment. If no host option is provided; it will default to the machine the code is being executed on.

We also need to add the inherited methods to the **EnvironmentCommand** class:

```
protected function getHost()
{
    return gethostname();
}

protected function getEnvironment()
{
    return App::environment();
}

protected function getPath()
```

```
12  {
13      return app_path() . "/config/environment.php";
14  }
15
16  protected function getConfig()
17  {
18      $environments = require $this->getPath();
19
20      if (!is_array($environments))
21      {
22          $environments = [];
23      }
24
25      return $environments;
26  }
27
28  protected function setConfig($config)
29  {
30      $config = "<?php return " . var_export($config, true) . ";";
31      File::put($this->getPath(), $config);
32  }
```

This was extracted from **app/commands/EnvironmentCommand.php**.

The **getConfig()** method fetches the contents of the **app/config/environment.php** file (a list of hosts per environment) and the **setConfig()** method writes back to it. We use the **var_export()** to re-create the array that's stored in memory; but it's possible to get a more aesthetically-pleasing configuration file. I've customised the **setConfig()** method to match my personal taste:

```
1   protected function setConfig($config)
2   {
3
4       $code = "<?php\n\nreturn [";
5
6       foreach ($config as $environment => $hosts)
7       {
8           $code .= "\n \"" . $environment . "\" => [";
9
10          foreach ($hosts as $host)
11          {
12              $code .= "\n \"" . $host . "\",";
13          }
14
```

```
15          $code = trim($code, ",");
16          $code .= "\n ],";
17      }
18
19      $code = trim($code, ",");
20      File::put($this->getPath(), $code . "\n];");
21  }
```

> This was extracted from **app/commands/EnvironmentCommand.php**.

In order for these environments to be of any use to us; we need to replace those defined in **bootstrap/start.php** with the following lines:

```
1  $env = $app->detectEnvironment(
2      require __DIR__ . "/../app/config/environment.php"
3  );
```

> This was extracted from **bootstrap/start.php**.

This ensures that the environments we set (using our environment commands), and not those hard-coded in the **bootstrap/start.php** file are used in determining the current machine environment.

Unsetting Environments

The last environment command we will create will provide us a way to remove hosts from an environment (in much the same way as they were added):

```php
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class EnvironmentRemoveCommand
extends EnvironmentCommand
{
    protected $name = "environment:remove";

    protected $description = "Removes host from environment.";

    public function fire()
    {
        $host        = $this->getHost();
        $config      = $this->getConfig();
        $overwrite   = $this->option("host");
        $environment = $this->argument("environment");

        if (!isset($config[$environment]))
        {
            $config[$environment] = [];
        }

        $use = $host;

        if ($overwrite)
        {
            $use = $overwrite;
        }

        foreach ($config[$environment] as $index => $item)
        {
            if ($item == $use)
            {
                unset($config[$environment][$index]);
            }
        }

        $this->setConfig($config);

```

```
43          $this->line(trim("
44              <info>Removed</info>
45              <comment>" . $use . "</comment>
46              <info>from</info>
47              <comment>" . $environment . "</comment>
48              <info>environment.</info>
49          "));
50      }
51
52      protected function getArguments()
53      {
54          return [
55              [
56                  "environment",
57                  InputArgument::REQUIRED,
58                  "Environment to remove the host from."
59              ]
60          ];
61      }
62
63      protected function getOptions()
64      {
65          return [
66              [
67                  "host",
68                  null,
69                  InputOption::VALUE_OPTIONAL,
70                  "Host to remove.",
71                  null
72              ]
73          ];
74      }
75  }
```

This file should be saved as **app/commands/EnvironmentRemoveCommand.php**.

It's pretty much the same as the **EnvironmentSetCommand** class, but instead of adding them to the configuration file; we remove them from the configuration file. It uses the same formatter as the **EnvironmentSetCommand** class.

> Commands make up a lot of this tutorial. It may, therefore, surprise you to know that there's not much more to them than this. Sure, there are different types (and values) of **InputOption**'s and **InputArgument**'s, but we're not going into that level of detail here.
>
> You can find out more about these at: **http://symfony.com/doc/current/components/console/introdu**

Asset Commands

There are two parts to managing assets. The first is how they are stored and referenced in views, and the second is how they are combined/minified.

Both of these operations will require a sane method of specifying asset containers and the assets contained therein. For this; we will create a new configuration file (in each environment we will be using):

```
1  <?php
2
3  return [
4      "header-css" => [
5          "css/bootstrap.css",
6          "css/shared.css"
7      ],
8      "footer-js" => [
9          "js/jquery.js",
10         "js/bootstrap.js",
11         "js/shared.js"
12     ]
13 ];
```

> This file should be saved as **app/config/local/asset.php**.

```php
<?php

return [
    "header-css" => [
        "css/shared.min.css" => [
            "css/bootstrap.css",
            "css/shared.css"
        ]
    ],
    "footer-js" => [
        "js/shared.min.js" => [
            "js/jquery.js",
            "js/bootstrap.js",
            "js/shared.js"
        ]
    ]
];
```

This file should be saved as **app/config/production/asset.php**.

The difference between these environment-based asset configuration files is how the files are combined in the production environment vs. how they are simply listed in the local environment. This makes development easier because you can see unaltered files while still developing and testing; while the production environment will have smaller file sizes.

To use these in our views; we're going to make a form macro. It's not technically what form macros were made for (since we're not using them to make forms) but it's such a convenient/clean method for generating view markup that we're going to use it anyway.

```php
<?php

Form::macro("assets", function($section)
{
    $markup = "";
    $assets = Config::get("asset");

    if (isset($assets[$section]))
    {
        foreach ($assets[$section] as $key => $value)
```

```
        {
            $use = $value;

            if (is_string($key))
            {
                $use = $key;
            }

            if (ends_with($use, ".css"))
            {
                $markup .= "<link
                    rel='stylesheet'
                    type='text/css'
                    href='" . asset($use) . "'
                />";
            }

            if (ends_with($use, ".js"))
            {
                $markup .= "<script
                    type='text/javascript'
                    src='" . asset($use) . "'
                ></script>";
            }
        }
    }

    return $markup;
});
```

> This file should be saved as **app/macros.php**.

This macro accepts a single parameter—the name of the section—and renders HTML markup elements for each asset file. It doesn't matter if there are a combination of stylesheets and scripts as the applicable tag is rendered for each asset type.

We also need to make sure this file gets included in our application's startup processes:

```
require app_path() . "/macros.php";
```

> This was extracted from **app/start/global.php**.

We can now use this in our templates...

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Laravel 4 - Deployment Tutorial</title>
        {{ Form::assets("header-css") }}
    </head>
    <body>
        <h1>Hello World!</h1>
        {{ Form::assets("footer-js") }}
    </body>
</html>
```

> This file should be saved as **app/views/hello.blade.php**.

Now, depending on the environment we have set; we will either see a list of asset files in each section, or single (production) asset files.

> You will notice the difference by changing your environment from local to production. Give the environment commands a go—this is the kind of thing they were made for!

Combining Assets

The simplest asset operation is combining. This is where we put two or more stylesheets or scripts together into a single file. Similarly to how we arranged the environment classes; the asset commands will inherit form a master command:

```php
<?php

use Assetic\Asset\AssetCollection;
use Assetic\Asset\FileAsset;
use Illuminate\Console\Command;

class AssetCommand
extends Command
{
    protected $name = "asset";

    protected $description = "Lists asset commands.";

    public function fire()
    {
        $this->line(trim("
            <comment>asset:combine</comment>
            <info>combines resource files.</info>
        "));

        $this->line(trim("
            <comment>asset:minify</comment>
            <info>minifies resource files.</info>
        "));
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [];
    }

    protected function getPath()
    {
        return public_path();
    }

    protected function getCollection($input, $filters = [])
```

```
43      {
44          $path       = $this->getPath();
45          $input      = explode(",", $input);
46          $collection = new AssetCollection([], $filters);
47
48          foreach ($input as $asset)
49          {
50              $collection->add(
51                  new FileAsset($path . "/" . $asset)
52              );
53          }
54
55          return $collection;
56      }
57
58      protected function setOutput($file, $content)
59      {
60          $path = $this->getPath();
61          return File::put($path . "/" . $file, $content);
62      }
63  }
```

This file should be saved as **app/commands/AssetCommand.php**.

Here's where we make use of Assetic. Among the many utilities Assetic provides; the **AssetCollection** and **FileAsset** classes will be out main focus. The **getCollection()** method accepts a comma-delimited list of asset files (relative to the **public** folder) and returns a populated **AssetCollection** instance.

```
1   <?php
2
3   use Illuminate\Console\Command;
4   use Symfony\Component\Console\Input\InputArgument;
5   use Symfony\Component\Console\Input\InputOption;
6
7   class AssetCombineCommand
8   extends AssetCommand
9   {
10      protected $name = "asset:combine";
```

```
11
12      protected $description = "Combines resource files.";
13
14      public function fire()
15      {
16          $input    = $this->argument("input");
17          $output   = $this->option("output");
18          $combined = $this->getCollection($input)->dump();
19
20          if ($output)
21          {
22              $this->line(trim("
23                  <info>Successfully combined</info>
24                  <comment>" . $input . "</comment>
25                  <info>to</info>
26                  <comment>" . $output . "</comment>
27                  <info>.</info>
28              "));
29
30              $this->setOutput($output, $combined);
31          }
32          else
33          {
34              $this->line($combined);
35          }
36      }
37
38      protected function getArguments()
39      {
40          return [
41              [
42                  "input",
43                  InputArgument::REQUIRED,
44                  "Names of input files."
45              ]
46          ];
47      }
48
49      protected function getOptions()
50      {
51          return [
52              [
```

```
                "output",
                null,
                InputOption::VALUE_OPTIONAL,
                "Name of output file.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/AssetCombineCommand.php**.

The **AssetCombineCommand** class expects the aforementioned list of assets and will output the results to console, by default. If we want to save the results to a file we can provide the **–output** option with a path to the combined file.

You can learn more about Assetic at: **https://github.com/kriswallsmith/assetic/**

Minifying Assets

Minifying asset files is just as easy; thanks to the Assetic-Minifier filters. Assetic provides filters which can be used to transform the output of asset files.

Usually some of these filters depend on third-party software being installed on the server, or things like Java applets. Fortunately, the Assetic-Minifier library provides pure PHP alternatives to CssMin and JSMin filters (which would otherwise need additional software installed).

```php
<?php

use Minifier\MinFilter;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class AssetMinifyCommand
extends AssetCommand
{
    protected $name = "asset:minify";

    protected $description = "Minifies resource files.";

    public function fire()
    {
        $type    = $this->argument("type");
        $input   = $this->argument("input");
        $output  = $this->option("output");
        $filters = [];

        if ($type == "css")
        {
            $filters[] = new MinFilter("css");
        }

        if ($type == "js")
        {
            $filters[] = new MinFilter("js");
        }

        $collection = $this->getCollection($input, $filters);
        $combined   = $collection->dump();

        if ($output)
        {
            $this->line(trim("
                <info>Successfully minified</info>
                <comment>" . $input . "</comment>
                <info>to</info>
                <comment>" . $output . "</comment>
                <info>.</info>
```

```
            "));

            $this->setOutput($output, $combined);
        }
        else
        {
            $this->line($combined);
        }
    }

    protected function getArguments()
    {
        return [
            [
                "type",
                InputArgument::REQUIRED,
                "Code type."
            ],
            [
                "input",
                InputArgument::REQUIRED,
                "Names of input files."
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "output",
                null,
                InputOption::VALUE_OPTIONAL,
                "Name of output file.",
                null
            ]
        ];
    }
}
```

> This file should be saved as **app/commands/AssetMinifyCommand.php**.

The minify command accepts two arguments—the type of assets (either **css** or **js**) and the comma-delimited list of asset files to minify. Like the combine command; it will output to console by default, unless the —output option is specified.

> You can learn more about Assetic-Minifier at: **https://github.com/toopay/assetic-minifier**

Building Assets

Commands that accept inputs and outputs are really useful for the combining and minifying asset files, but it's a pain to have to specify inputs and outputs each time (especially when we have gone to the effort of defining asset lists in configuration files). For this purpose; we need a command which will combine and minify all the asset files appropriately.

```
1  <?php
2
3  use Illuminate\Console\Command;
4
5  class BuildCommand
6  extends Command
7  {
8      protected $name = "build";
9
10     protected $description = "Builds resource files.";
11
12     public function fire()
13     {
14         $sections = Config::get("asset");
15
16         foreach ($sections as $section => $assets)
17         {
18             foreach ($assets as $output => $input)
19             {
20                 if (!is_string($output))
```

```
21                  {
22                      continue;
23                  }
24
25                  if (!is_array($input))
26                  {
27                     $input = [$input];
28                  }
29
30                  $input = join(",", $input);
31
32                  $options = [
33                      "--output" => $output,
34                      "input"    => $input
35                  ];
36
37                  if (ends_with($output, ".min.css"))
38                  {
39                      $options["type"] = "css";
40                      $this->call("asset:minify", $options);
41                  }
42                  else if (ends_with($output, ".min.js"))
43                  {
44                      $options["type"] = "js";
45                      $this->call("asset:minify", $options);
46                  }
47                  else
48                  {
49                      $this->call("asset:combine", $options);
50                  }
51              }
52          }
53      }
54  }
```

This file should be saved as **app/commands/BuildCommand.php**.

The **BuildCommand** class fetches the asset lists, from the configuration files, and iterates over them; combining/minifying as needed. It does this by checking file extensions. If the file ends in **.min.js**

then the minify command is run in js mode. Files ending in **.min.css** are minified in css mode, and so forth.

The **app/config/*/asset.php** files are environment-based. This means running the build command will build the assets for the current environment. Often this will not be the environment you want to build assets for. I often build assets on my local machine, yet I want assets built for production. When that is the case; I provide the **–env=production** flag to the build command.

Watching Assets

So we have the tools to combine, minify and even build our asset files. It's a pain to have to remember to do those things all the time (especially when files are being updated at a steady pace), so we're going to take it a step further by adding a file watcher.

Remember the Resource Watcher library we added to **composer.json**? Well we need to add it to the list of service providers:

```
"providers" => [
    // other service providers here
   "JasonLewis\ResourceWatcher\Integration\LaravelServiceProvider"
]
```

> This was extracted from **app/config/app.php**.

The Resource Watcher library requires Laravel 4's **Filesystem** library, and this service provider will allow us to utilise dependency injection.

```
<?php

use Illuminate\Console\Command;

class WatchCommand
extends Command
{
    protected $name = "watch";

    protected $description = "Watches for file changes.";

    public function fire()
```

```
    {
        $path     = $this->getPath();
        $watcher  = App::make("watcher");
        $sections = Config::get("asset");

        foreach ($sections as $section => $assets)
        {
            foreach ($assets as $output => $input)
            {
                if (!is_string($output))
                {
                    continue;
                }

                if (!is_array($input))
                {
                    $input = [$input];
                }

                foreach ($input as $file)
                {
                    $watch    = $path . "/" . $file;
                    $listener = $watcher->watch($watch);

                    $listener->onModify(function() use (
                        $section,
                        $output,
                        $input,
                        $file
                    )
                    {
                        $this->build(
                            $section,
                            $output,
                            $input,
                            $file
                        );
                    });
                }
            }
        }

```

```
55          $watcher->startWatch();
56      }
57
58      protected function build($section, $output, $input, $file)
59      {
60          $options = [
61              "--output" => $output,
62              "input"    => join(",", $input)
63           ];
64
65           $this->line(trim("
66              <info>Rebuilding</info>
67              <comment>" . $output . "</comment>
68              <info>after change to</info>
69              <comment>" . $file . "</comment>
70              <info>.</info>
71           "));
72
73           if (ends_with($output, ".min.css"))
74           {
75              $options["type"] = "css";
76              $this->call("asset:minify", $options);
77           }
78           else if (ends_with($output, ".min.js"))
79           {
80              $options["type"] = "js";
81              $this->call("asset:minify", $options);
82           }
83           else
84           {
85              $this->call("asset:combine", $options);
86           }
87      }
88
89      protected function getArguments()
90      {
91          return [];
92      }
93
94      protected function getOptions()
95      {
96          return [];
```

```
97      }
98
99      protected function getPath()
100     {
101         return public_path();
102     }
103 }
```

> This file should be saved as **app/commands/WatchCommand.php**.

The **WatchCommand** class is a step up from the **BuildCommand** class in that it processes asset files similarly. Where it excels is in how it is able to watch the individual files in **app/config/*/asset.php**.

When a **Watcher** targets a specific file (with the **$watcher->watch()** method); it generates a **Listener**. Listeners can have events bound to them (as is the case with the **onModify()** event listener method).

When these files change, the processed asset files they form part of are rebuilt, using the same logic as in the build command.

> You have to have the watcher running in order for updates to cascade into combined/minified asset files. Do this by running: **php artisan watch**
>
> You can learn more about the Resource Watcher library at: **https://github.com/jasonlewis/resource-watcher**

Resource Watcher Integration Bug

While creating this tutorial; I found a bug with the service provider. It relates to class resolution, and was probably caused by a reshuffling of the library. I have since contacted Jason Lewis to inform him of the bug, and submitted a pull request which resolves it.

If you are having issues relating to the Resource Watcher library, try replacing the service provider file with this one:

```php
<?php namespace JasonLewis\ResourceWatcher\Integration;

use Illuminate\Support\ServiceProvider;
use JasonLewis\ResourceWatcher\Tracker;
use JasonLewis\ResourceWatcher\Watcher;

class LaravelServiceProvider extends ServiceProvider {

    /**
    * Indicates if loading of the provider is deferred.
    *
    * @var bool
    */
    protected $defer = false;

    /**
    * Register the service provider.
    *
    * @return void
    */
    public function register()
    {
        $this->app['watcher'] = $this->app->share(function($app)
        {
            $tracker = new Tracker;
            return new Watcher($tracker, $app['files']);
        });
    }

    /**
    * Get the services provided by the provider.
    *
    * @return array
    */
    public function provides()
    {
        return array('watcher');
    }
}
```

> This file should be saved as **vendor/jasonlewis/resource-watcher/src/JasonLewis/ResourceWatcher/Integration/LaravelServiceProvider.php**.

Rsync

Rsync is a file synchronisation utility which we will use to synchronise our distribution folder to a remote server. It requires a valid private/public key and some configuration.

To set up SSH access, for your domain, follow these steps:

1. Back up any keys you already have in ~/**.ssh**
2. Generate a new key with: **ssh-keygen -t rsa -C "your_email@example.com"**
3. Remember the name you set here.
4. Copy the contents of the new public key file (the name from step 3, ending in **.pub**).
5. SSH into your remove server.
6. Add the contents of the new public key file (which you copied in step 3) to ~/**.ssh/authorized_keys**. Add the following lines to ~/**.ssh/config** (on your local machine):

```
host example.com
    User your_user
    IdentityFile your_key_file
```

> This was extracted from ~/**.ssh/config**.

> The **your_user** account needs to be the same as the one with which you SSH'd into the remote server and added the authorised key.
>
> The **your_identity_file** is the name from step 3 (not ending in **.pub**).

Now, when you type **ssh example.com** (where **example.com** is the name of the domain you've been accessing with SSH); you should be let in without even having to provide a password. Don't worry—your server is still secure. You've just let it know (ahead of time) what an authentic connection from you looks like.

With this configuration in place; you won't need to do anything tricky in order to get Rsync to work correctly. The hard part is getting a good connection…

Distribute Command

In order for us to distribute our code, we need to make a copy of it and perform some operations on the copy. This involves optimisation and cleanup.

Copying Files For Distribution

First, let's make the copy command:

```
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;

class CopyCommand
extends Command
{
    protected $name = "copy";

    protected $description = "Creates distribution files.";

    public function fire()
    {
        $target = $this->option("target");

        if (!$target)
        {
            $target = "../distribution";
        }

        File::copyDirectory("./", $target);

        $this->line(trim("
            <info>Successfully copied source files to</info>
```

```
26              <comment>" . realpath($target) . "</comment>
27              <info>.</info>
28          "));
29      }
30
31      protected function getArguments()
32      {
33          return [];
34      }
35
36      protected function getOptions()
37      {
38          return [
39              [
40                  "target",
41                  null,
42                  InputOption::VALUE_OPTIONAL,
43                  "Distribution path.",
44                  null
45              ]
46          ];
47      }
48  }
```

> This file should be saved as **app/commands/CopyCommand.php**.

The copy command uses Laravel 4's **File** methods to copy the source directly recursively. It initially targets **../distribute** but this can be changed with the **–target** option.

> It's important that you copy the distribution files to a target outside of your source folder. The copy command copies ./ which means a target inside will lead to an "infinite copy loop" where the command tries to copy the distribution folder into itself an infinite number of times.
>
> To get around this; I guess you could target sources files individually (or in folders). It's likely that you will be running the copy command from a local machine, so there's little harm in copying the distribution files into a directory outside of the one your application files are in.

Removing Development Files

Next up, we need a command to remove any temporary/development files. We'll start by creating a config file in which these files are listed:

```
<?php

return [
    "app/storage/cache/*",
    "app/storage/logs/*",
    "app/storage/sessions/*",
    "app/storage/views/*",
    ".DS_Store",
    ".git*",
    ".svn*",
    "public/js/jquery.js",
    "public/js/bootstrap.js",
    "public/js/shared.js",
    "public/css/bootstrap.css",
    "public/css/shared.css"
];
```

> This file should be saved as **app/config/clean.php**.

I've just listed a few common temporary/development files. You can use any syntax that works with PHP's **glob()** function, as this is what Laravel 4 is using behind the scenes.

I'm also removing the development scripts and styles, as these will be built into minified asset files by the build command.

```
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;

class CleanCommand
extends Command
{
    protected $name = "clean";
```

```
10
11      protected $description = "Cleans distribution folder.";
12
13      public function fire()
14      {
15          $target = $this->option("target");
16
17          if (!$target)
18          {
19              $target = "../distribution";
20          }
21
22          $cleaned = Config::get("clean");
23
24          foreach ($cleaned as $pattern)
25          {
26              $paths = File::glob($target . "/" . $pattern);
27
28              foreach ($paths as $path)
29              {
30                  if (File::isFile($path))
31                  {
32                      File::delete($path);
33                  }
34
35                  if (File::isDirectory($path))
36                  {
37                      File::deleteDirectory($path);
38                  }
39              }
40
41              $this->line(trim("
42                  <info>Deleted all files/folders matching</info>
43                  <comment>" . $pattern . "</comment>
44                  <info>.</info>
45              "));
46          }
47      }
48
49      protected function getArguments()
50      {
51          return [];
```

```
52      }
53
54      protected function getOptions()
55      {
56          return [
57              [
58                  "target",
59                  null,
60                  InputOption::VALUE_OPTIONAL,
61                  "Distribution path.",
62                  null
63              ]
64          ];
65      }
66  }
```

> This file should be saved as **app/commands/CleanCommand.php**.

The **CleanCommand** class gets all files matching the patterns in **app/config/clean.php** and deletes them. It handles folders as well.

> Be very careful when deleting files. It's always advisable to make a full backup before you test these kinds of scripts. I nearly nuked all the tutorial code because I didn't make a backup!

Synchronising Files To A Remote Server

We've set up SSH access and we have code ready for deployment. Before we sync, we should set up a config file for target remote servers:

```
<?php

return [
    "production" => [
        "url"  => "example.com",
        "path" => "/var/www"
    ]
];
```

> This file should be saved as **app/config/host.php**.

The normal SSH access we've set up takes care of the username and password, so all we need to be able to configure is the url of the remote server and the path on it to upload the files to.

```
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class DistributeCommand
extends Command
{
    protected $name = "distribute";

    protected $description = "Synchronises files with target.";

    public function fire()
    {
        $host   = $this->argument("host");
        $target = $this->option("target");

        if (!$target)
        {
            $target = "../distribution";
        }

        $url  = Config::get("host." . $host . ".url");
        $path = Config::get("host." . $host . ".path");
```

```
26
27          $command = "rsync --verbose --progress --stats --compress --recursive --t\
28  imes --perms -e ssh " . $target . "/ " . $url . ":" . $path . "/";
29
30          $escaped = escapeshellcmd($command);
31
32          $this->line(trim("
33              <info>Synchronizing distribution files to</info>
34              <comment>" . $host . " (" . $url . ")</comment>
35              <info>.</info
36          "));
37
38          exec($escaped, $output);
39
40          foreach ($output as $line)
41          {
42              if (starts_with($line, "Number of files transferred"))
43              {
44                  $parts = explode(":", $line);
45
46                  $this->line(trim("
47                      <comment>" . trim($parts[1]) . "</comment>
48                      <info>files transferred.</info>
49                  "));
50               }
51
52              if (starts_with($line, "Total transferred file size"))
53              {
54                  $parts = explode(":", $line);
55                  $this->line(trim("
56                      <comment>" . trim($parts[1]) . "</comment>
57                      <info>transferred.</info>
58                  "));
59              }
60          }
61      }
62
63      protected function getArguments()
64      {
65          return [
66              [
67                  "host",
```

```
68                  InputArgument::REQUIRED,
69                  "Destination host."
70              ]
71          ];
72      }
73
74      protected function getOptions()
75      {
76          return [
77              [
78                  "target",
79                  null,
80                  InputOption::VALUE_OPTIONAL,
81                  "Distribution path.",
82                  null
83              ]
84          ];
85      }
86  }
```

This file should be saved as **app/commands/DistributeCommand.php**.

The **DistributeCommand** class accepts a host (remote server name) argument and a **–target** option. Like the copy and clean commands; the **–target** option will override the default **../distribute** folder with one of your choosing.

The host argument needs to match a key in your **app/config/host.php** configuration file. It carefully constructs and escapes a shell command instruction rsync to synchronise files from the distribution folder to a path on the remote server.

Once the files have been synchronised, it will inspect the messy output of the rsync command and return number number of bytes and files transferred.

The intricacies of SSH and Rsync are outside the scope of this tutorial. You can learn more about Rsync at: **https://calomel.org/rsync_tips.html**

You can watch the progress of an upload (to the remote server) by connecting via SSH, navigating to the folder and running the command: **du -hs**

Command Portability

Portability refers to how easily code will run on any environment. Most of the commands created in this tutorial are portable. The parts that aren't too portable are those relating to Rsync. Rsync is a *nix utility, and is therefore not found on Windows machines.

If you find the deploy command gives you headaches; feel free to jump in just before it and deploy the distribution folder (after build, copy and clean commands) by whatever means works for you.

Preprocessors

Preprocessors are things which convert some intermediate languages into common languages (such as converting Less, SASS, Stylus into CSS). These are often combined with deployment workflows.

Due to time constraints; I've not covered them in this tutorial. If you need to add them then a good place would be as a filter in the **asset:combine** or **asset:minify** commands. You may also want to add another command (to be executed before those two) which preprocesses any of these languages.

Images

Image optimisation is a huge topic. I might go into detail on how to optimise your images (as part of the deployment process) in a future tutorial. Assetic does provide filters for this sort of thing; so check its documentation if you feel up to the challenge!

API

I seldom think of MVC in terms of applications which don't really have views. Turns out Laravel 4 is stocked with features which make REST API's a breeze to create and maintain.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-api**

Dependencies

One of the goals, of this tutorial, is to speed up our prototyping. To that end; we'll be installing Jeffrey Way's Laravel 4 Generators. This can be done by amending the **composer.json** file to including the following dependency:

```
1  "way/generators" : "dev-master"
```

> This was extracted from **composer.json**.

We'll also need to add the **GeneratorsServiceProvider** to our app config:

```
1  "Way\Generators\GeneratorsServiceProvider"
```

> This was extracted from **app/config/app.php**.

You should now see the generate methods when you run artisan.

Creating Resources With Artisan

Artisan has a few tasks which are helpful in setting up resourceful API endpoints. New controllers can be created with the command:

```
php artisan controller:make EventController
```

New migrations can also be created with the command:

```
php artisan migrate:make CreateEventTable
```

These are neat shortcuts but they're a little limited considering our workflow. What would make us even more efficient is if we had a way to also generate models and seeders. Enter Jeffrey Way's Laravel 4 Generators...

> You can learn more about Laravel 4's built-in commands by running php artisan in your console, or at: **http://laravel.com/docs**.

Creating Resources With Generators

With the generate methods installed; we can now generate controllers, migrations, seeders and models.

Generating Migrations

Let's begin with the migrations:

```
php artisan generate:migration create_event_table
```

This simple command will generate the following migration code:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateEventTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
```

```
12      */
13      public function up()
14      {
15          Schema::create('event', function(Blueprint $table) {
16              $table->increments('id');
17
18              $table->timestamps();
19          });
20      }
21
22      /**
23      * Reverse the migrations.
24      *
25      * @return void
26      */
27      public function down()
28      {
29          Schema::drop('event');
30      }
31  }
```

> This file should be saved as **app/database/migrations/0000*00*00_000000_create_event_table.php**.

We've seen these kinds of migrations before, so there's not much to say about this one. The generators allow us to take it a step further by providing field names and types:

```
1  php artisan generate:migration --fields="name:string, description:text, started_a\
2  t:timestamp, ended_at:timestamp" create_event_table
```

This command alters the **up()** method previously generated:

```
public function up()
{
    Schema::create('event', function(Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->text('description');
        $table->timestamp('started_at');
        $table->timestamp('ended_at');
        $table->timestamps();
    });
}
```

> This was extracted from **app/database/migrations/0000*000*000_0000000_create_event_table.php**.

Similarly, we can create tables for sponsors and event categories:

```
php artisan generate:migration --fields="name:string, description:text" create_ca\
tegory_table

php artisan generate:migration --fields="name:string, url:string, description:tex\
t" create_sponsor_table
These commands generate the following migrations:
```

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateCategoryTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
```

```
        Schema::create('category', function(Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->text('description');
            $table->timestamps();
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('category');
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_create_category_table.php**.

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateSponsorTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('sponsor', function(Blueprint $table) {
            $table->increments('id');
```

```
            $table->string('name');
            $table->string('url');
            $table->text('description');
            $table->timestamps();
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('sponsor');
    }
}
```

> This file should be saved as **app/database/migrations/0000*00*00_000000_create_sponsor_table.php**.

The last couple of migrations we need to create are for pivot tables to connect sponsors and categories to events. Pivot tables are common in HABTM (Has And Belongs To Many) relationships, between database entities.

The command for these is just as easy:

```
php artisan generate:pivot event category

php artisan generate:pivot event sponsor
These commands generate the following migrations:
```

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class PivotCategoryEventTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('category_event', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('category_id')->unsigned()->index();
            $table->integer('event_id')->unsigned()->index();
            $table->foreign('category_id')->references('id')->on('category')->onD\
elete('cascade');
            $table->foreign('event_id')->references('id')->on('event')->onDelete(\
'cascade');
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('category_event');
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_pivot_category_-event_table.php**.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class PivotEventSponsorTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('event_sponsor', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('event_id')->unsigned()->index();
            $table->integer('sponsor_id')->unsigned()->index();
            $table->foreign('event_id')->references('id')->on('event')->onDelete('\
'cascade');
            $table->foreign('sponsor_id')->references('id')->on('sponsor')->onDel\
ete('cascade');
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('event_sponsor');
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_pivot_event_sponsor_-table.php**.

> Pay special attention to the names of the tables in the calls to the **on()** method. I have changed them from plural tables names to singular table names. The templates used to construct these constraints don't seem to take into account the name of the table on which the constrains are being created.

Apart from the integer fields (which we've seen before); these pivot tables also have foreign keys, with constraints. These are common features of relational databases, as they help to maintain referential integrity among database entities.

> These migrations specify foreign key constraints. This means the database will require valid foreign keys when you insert and update rows in these tables. Your seeder class needs to provide these valid ID values, or you should remove the constraints (**references('id')**...**onDelete('cascade')**).

With all these migration files created; we have only to migrate them to the database with:

```
1 php artisan migrate
```

(This assumes you have already configured the database connection details, in the configuration files.)

Generating Seeders

Seeders are next on our list. These will provide us with starting data; so our API responses don't look so empty.

```
1 php artisan generate:seed Category
2 php artisan generate:seed Sponsor
```

These commands will generate stub seeders, and add them to the **DatabaseSeeder** class. I have gone ahead and customised them to include some data (and better formatting):

```php
<?php

class CategoryTableSeeder
extends Seeder
{
    public function run()
    {
        DB::table("category")->truncate();

        $categories = [
            [
                "name"        => "Concert",
                "description" => "Music for the masses.",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "Competition",
                "description" => "Prizes galore.",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "General",
                "description" => "Things of interest.",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ]
        ];

        DB::table("category")->insert($categories);
    }
}
```

This file should be saved as **app/database/seeds/CategoryTableSeeder.php**.

```php
<?php

class SponsorTableSeeder
extends Seeder
{
    public function run()
    {
        DB::table("sponsor")->truncate();

        $sponsors = [
            [
                "name"        => "ACME",
                "description" => "Makers of quality dynomite.",
                "url"         => "http://www.kersplode.com",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "Cola Company",
                "description" => "Making cola like no other.",
                "url"         => "http://www.cheerioteeth.com",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "MacDougles",
                "description" => "Super sandwiches.",
                "url"         => "http://www.imenjoyingit.com",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ]
        ];

        DB::table("sponsor")->insert($sponsors);
    }
}
```

This file should be saved as **app/database/seeds/SponsorTableSeeder.php**.

To get this data into the database, we need to run the seed command:

```
php artisan db:seed
```

> If you kept the foreign key constraints (in the pivot table migrations); you will need to modify your seeder classes to include valid foreign keys.

At this point; we should have the database tables set up, and some test data should be in the category and sponsor tables.

Generating Models

Before we can output data, we need a way to interface with the database. We're going to use models for this purpose, so we should generate some:

```
php artisan generate:model Event

php artisan generate:model Category

php artisan generate:model Sponsor
```

These commands will generate stub models which resemble the following:

```
<?php

class Event extends Eloquent {

    protected $guarded = array();

    public static $rules = array();
}
```

> This file should be saved as **app/models/Event.php**.

We need to clean these up a bit, and add the relationship data in...

```php
<?php

class Event
extends Eloquent
{
    protected $table = "event";

    protected $guarded = [
        "id",
        "created_at",
        "updated_at"
    ];

    public function categories()
    {
        return $this->belongsToMany("Category", "category_event", "event_id", "ca\
tegory_id");
    }

    public function sponsors()
    {
        return $this->belongsToMany("Sponsor", "event_sponsor", "event_id", "spon\
sor_id");
    }
}
```

> This file should be saved as **app/models/Event.php**.

```php
<?php

class Category
extends Eloquent
{
    protected $table = "category";

    protected $guarded = [
        "id",
        "created_at",
```

```
        "updated_at"
    ];

    public function events()
    {
        return $this->belongsToMany("Event", "category_event", "category_id", "ev\
ent_id");
    }
}
```

This file should be saved as **app/models/Category.php**.

```
<?php

class Sponsor
extends Eloquent
{
    protected $table = "sponsor";

    protected $guarded = [
        "id",
        "created_at",
        "updated_at"
    ];

    public function events()
    {
        return $this->belongsToMany("Event", "event_sponsor", "sponsor_id", "even\
t_id");
    }
}
```

This file should be saved as **app/models/Sponsor.php**.

As I mentioned before; we've gone with a **belongsToMany()** relationship to connect the entities together. The arguments for each of these is (1) the model name, (2) the pivot table name, (3) the local key and (4) the foreign key.

> I refer to them as the local and foreign keys but they are actually both foreign keys on the pivot table. Think of local as the key closest to the model in which the relationship is defined and the foreign as the furthest.

> Our models' $table property should match what's specified in the migrations and seeders.

The last step in creating our API is to create the client-facing controllers.

Generating Controllers

The API controllers are different from those you might typically see, in a Laravel 4 application. They don't load views; rather they respond to the requested content type. They don't typically cater for multiple request types within the same action. They are not concerned with interface; but rather translating and formatting model data.

Creating them is a bit more tricky than the other classes we've done so far:

```
1  php artisan generate:controller EventController
```

The command isn't much different, but the generated file is far from ready:

```
1  <?php
2  class EventController extends BaseController {
3
4      /**
5      * Display a listing of the resource.
6      *
7      * @return Response
8      */
9      public function index()
10     {
11         return View::make('events.index');
```

```
12      }
13
14      /**
15      * Show the form for creating a new resource.
16      *
17      * @return Response
18      */
19      public function create()
20      {
21          return View::make('events.create');
22      }
23
24      /**
25      * Store a newly created resource in storage.
26      *
27      * @return Response
28      */
29      public function store()
30      {
31          //
32      }
33
34      /**
35      * Display the specified resource.
36      *
37      * @param int $id
38      * @return Response
39      */
40      public function show($id)
41      {
42          return View::make('events.show');
43      }
44
45      /**
46      * Show the form for editing the specified resource.
47      *
48      * @param int $id
49      * @return Response
50      */
51      public function edit($id)
52      {
53          return View::make('events.edit');
```

```
54      }
55
56      /**
57      * Update the specified resource in storage.
58      *
59      * @param int $id
60      * @return Response
61      */
62      public function update($id)
63      {
64          //
65      }
66
67      /**
68      * Remove the specified resource from storage.
69      *
70      * @param int $id
71      * @return Response
72      */
73      public function destroy($id)
74      {
75          //
76      }
77  }
```

This file should be saved as **app/controllers/EventController.php**.

We've not going to be rendering views, so we can remove those statements/actions. We're also not going to deal just with integer ID values (we'll get to the alternative shortly).

For now; what we want to do is list events, create them, update them and delete them. Our controller should look similar to the following:

```php
<?php

class EventController
extends BaseController
{
    public function index()
    {
        return Event::all();
    }

    public function store()
    {
        return Event::create([
            "name"        => Input::get("name"),
            "description" => Input::get("description"),
            "started_at"  => Input::get("started_at"),
            "ended_at"    => Input::get("ended_at")
        ]);
    }

    public function show($event)
    {
        return $event;
    }

    public function update($event)
    {
        $event->name        = Input::get("name");
        $event->description = Input::get("description");
        $event->started_at  = Input::get("started_at");
        $event->ended_at    = Input::get("ended_at");
        $event->save();
        return $event;
    }

    public function destroy($event)
    {
        $event->delete();
        return Response::json(true);
    }
}
```

This file should be saved as **app/controllers/EventController.php**.

We've deleted a bunch of actions and added some simple logic in others. Our controller will list (**index**) events, show them individually, create (**store**) them, update them and delete (**destroy**) them.

We need to return a JSON response differently, for the **destroy()** action, as boolean return values do not automatically map to JSON responses in the same way as collections and models do.

You can optionally specify the template to be used in the creation of seeders, models and controllers. you do this by providing the **–template** option with the path to a Mustache template file. Unfortunately migrations do not have this option. If you find yourself using these generators often enough, and wanting to customise the way in which new class are created; you will probably want to take a look at this option (and the template files in **vendor/way/generators/src/Way/Generators/Generators/templates**).

You can find out more about Jeffrey Way's Laravel 4 Generators at: **https://github.com/JeffreyWay/Laravel-4-Generators**.

Binding Models To Routes

Before we can access these; we need to add routes for them:

```
Route::model("event", "Event");

Route::get("event", [
    "as"   => "event/index",
    "uses" => "EventController@index"
]);

Route::post("event", [
    "as"   => "event/store",
    "uses" => "EventController@store"
]);

Route::get("event/{event}", [
    "as"   => "event/show",
    "uses" => "EventController@show"
]);

Route::put("event/{event}", [
    "as"   => "event/update",
    "uses" => "EventController@update"
]);

Route::delete("event/{event}", [
    "as"   => "event/destroy",
    "uses" => "EventController@destroy"
]);
```

> This was extracted from **app/routes.php**.

There are two important things here:

1. We're binding a model parameter to the routes. What that means is that we are telling Laravel to look for a specific parameter (in this case **event**) and treat the value found as an ID. Laravel will attempt to return a model instance if it finds that parameter in a route.
2. We're using different request methods to do perform different actions on our events. **GET** requests retrieve data, **POST** requests store data. **PUT** requests update data and **DELETE** requests delete data. This is called REST (Representational State Transfer).

> Laravel 4 does support the **PATCH** method, though it's undocumented. Many API's support/use the **PUT** method when what they implement is the closer to the **PATCH** method. That's a discussion for elsewhere; but just know you could use both without modification to Laravel 4.

> You can find out more about Route Model Binding at: **http://laravel.com/docs/routing#route-model-binding**.

Troubleshooting Aliases

If you go to the index route; you will probably see an error. It might say something like "Call to undefined method Illuminate\Events\Dispatcher::all()". This is because there is already a class (or rather an alias) called **Event**. **Event** is the name of our model, but instead of calling the **all()** method on our model; it's trying to call it on the event disputer class baked into Laravel 4.

I've lead us to this error intentionally, to demonstrate how to overcome it if you ever have collisions in your applications. Most everything in Laravel 4 is in a namespace. However, to avoid lots of extra keystrokes; Laravel 4 also offers a configurable list of aliases (in **app/config/app.php**).

In the list of aliases; you will see an entry which looks like this:

```
1  'Event' => 'Illuminate\Support\Facades\Event',
```

> This was extracted from **app/config/app.php**.

I changed the key of that entry to **Events** but you can really call it anything you like.

> It's often just easier to change the name of the classes you create than it is to alter the aliases array. There are many Laravel 4 tutorials that will refer to these classes by their alias, so you'll need to keep a mental log-book of the aliases you've changed and where they might be referred to.

Testing Endpoints

It's not always easy to test REST API's simply by using the browser. Often you will need to use an application to perform the different request methods. Thankfully modern *nix systems already have the Curl library, which makes these sorts of tests easier.

You can test the index endpoint with the console command:

```
curl http://dev.tutorial-laravel-4-api/event
```

> Your host (domain) name will differ based on your own setup.

> You should only be seeing a JSON response - if you're seeing HTML there's a good chance it's a Laravel error page. Check your application logs for more details.

Unless you've manually populated the event table, or set up a seeder for it; you should see an empty JSON array. This is a good thing, and also telling of some Laravel 4 magic. Our **index()** action returns a collection, but it's converted to JSON output when passed in a place where a Response is expected.

Let's add a new event:

```
curl -X POST -d "name=foo&description=a+day+of+foo&started_at=2013-10-03+09:00&en\
ded_at=2013-10-03+12:00" http://dev.tutorial-laravel-4-api:2080/event
```

..now, when we request the **index()** action, we should see the new event has been added. We can retrieve this event individually with a request similar to this:

```
curl http://dev.tutorial-laravel-4-api:2080/event/1
```

There's a lot going on here. Remember how we bound the model to a specific parameter name (in **app/routes.php**)? Well Laravel 4 sees that ID value, matches it to the bound model parameter and fetches the record from the database. If the ID does not match any of the records in the database; Laravel will respond with a 404 error message.

If the record is found; Laravel returns the model representation to the action we specified, and we get a model to work with.

Let's update this event:

```
curl -X PUT -d "name=best+foo&description=a+day+of+the+best+foo&started_at=2013-1\
0-03+10:00&ended_at=2013-10-03+13:00" http://dev.tutorial-laravel-4-api:2080/even\
t/1
```

Notice how all that's changed is the data and the request type—even though we're doing something completely different behind the scenes.

Lastly, let's delete the event:

```
curl -X DELETE http://dev.tutorial-laravel-4-api:2080/event/1
```

> Feel free to set the same routes and actions up for categories and sponsors. I've not covered them here, for the sake of time, but they only differ in terms of the fields.

Authenticating Requests

So far we've left the API endpoints unauthenticated. That's ok for internal use but it would be far more secure if we were to add an authentication layer.

We do this by securing the routes in filtered groups, and checking for valid credentials within the filter:

```
Route::group(["before" => "auth"], function()
{
    // ...routes go here
});
```

> This was extracted from **app/routes.php**.

```
Route::filter("auth", function()
{
    // ...get database user

    if (Input::server("token") !== $user->token)
    {
        App::abort(400, "Invalid token");
    }
});
```

> This was extracted from **app/filters.php**.

Your choice for authentication mechanisms will greatly affect the logic in your filters. I've opted not to go into great detail with regards to how the tokens are generated and users are stored. Ultimately; you can check for token headers, username/password combos or even IP addresses.

What's important to note here is that we check for this thing (tokens in this case) and if they do not match those stored in user records, we abort the application execution cycle with a 400 error (and message).

You can find out more about filters at: http://laravel.com/docs/routing#route-filters.

Using Accessors And Mutators

There are times when we need to customise how model attributes are stored and retrieved. Laravel 4 lets us do that by providing specially named methods for accessors and mutators:

```
public function setNameAttribute($value)
{
    $clean = preg_replace("/\W/", "", $value);
    $this->attributes["name"] = $clean;
}

public function getDescriptionAttribute()
{
    return trim($this->attributes["description"]);
}
```

> This was extracted from **app/models/Event.php**.

You can catch values, before they hit the database, by creating public **set*Attribute()** methods. These should transform the **$value** in some way and commit the change to the internal **$attributes** array.

You can also catch values, before they are returned, by creating **get*Attribute()** methods.

In the case of these methods; I am removing all non-word characters from the name value, before it hits the database; and trimming the description before it's returned by the property accessor. Getters are also called by the **toArray()** and **toJson()** methods which transform model instances into either arrays or JSON strings.

You can also add attributes to models by creating accessors and mutators for them, and mentioning them in the **$appends** property:

```
1  protected $appends = ["hasCategories", "hasSponsors"];
2
3  public function getHasCategoriesAttribute()
4  {
5      $hasCategories = $this->categories()->count() > 0;
6      return $this->attributes["hasCategories"] = $hasCategories;
7  }
8
9  public function getHasSponsorsAttribute()
10 {
11     $hasSponsors = $this->sponsors()->count() > 0;
12     return $this->attributes["hasSponsors"] = $hasSponsors;
13 }
```

> This was extracted from **app/models/Event.php**.

Here we've created two new accessors which check the count for categories and sponsors. We've also added those two attributes to the **$appends** array so they are returned when we list (**index**) all events or specific (**show**) events.

> You can find out more about attribute accessor and mutators at: **http://laravel.com/docs/eloquent#accessors-and-mutators**.

Using Cache

Laravel 4 provides a great cache mechanism. It's configured in the same was as the database:

```
<?php

return [
    "driver"    => "memcached",
    "memcached" => [
        [
            "host"   => "127.0.0.1",
            "port"   => 11211,
            "weight" => 100
        ]
    ],
    "prefix" => "laravel"
];
```

> This file should be saved as **app/config/cache.php**.

I've configured my cache to use the Memcached provider. This needs to be running on the specified host (at the specified port) in order for it to work.

> Installing and running Memcached are outside the scope of this tutorial. It's complicated.

No matter the provider you choose to use; the cache methods work the same way:

```
public function index()
{
    return Cache::remember("events", 15, function()
    {
        return Event::all();
    });
}
```

> This was extracted from **app/controllers/EventController.php**.

The **Cache::remember()** method will store the callback return value in cache if it's not already there. We've set it to store the events for 15 minutes.

The primary use for cache is in key/value storage:

```
public function total()
{
    if (($total = Cache::get("events.total")) == null)
    {
        $total = Event::count();
        Cache::put("events.total", $total, 15);
    }

    return Response::json((int) $total);
}
```

> This was extracted from **app/controllers/EventController.php**.

You can also invoke this cache on Query Builder queries or Eloquent queries:

```
public function today()
{
    return Event::where(DB::raw("DAY(started_at)"), date("d"))
        ->remember(15)
        ->get();
}
```

> This was extracted from **app/controllers/EventController.php**.

...we just need to remember to add the **remember()** method before we call the **get()** or **first()** methods.

> You can find out more about cache at: **http://laravel.com/docs/cache**.

Packages

Packages are the recommended way of extending the functionality provided by Laravel 4. They're nothing more than Composer libraries with some particular bootstrapping code.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-packages**

This chapter follows on from a previous chapter which covered the basics of creating a deployment process for your applications. It goes into detail about creating a Laravel 4 installation; so you should be familiar with it before spending a great deal of time in this one. You'll also find the source-code we created there to be a good basis for understanding the source code of this chapter.

I learned the really technical bits of this chapter from reading Taylor's book. If you take away just one thing from this it should be to get and read it.

Composer

While the previous chapter shows you how to create a Laravel 4 project; we need to do a bit more work in this area if we are to understand one of the fundamental ways in which packages differ from application code.

Laravel 3 had the concept of bundles; which were downloadable folders of source code. Laravel 4 extends this idea, but instead of rolling its own download process, it makes use of Composer. Composer should not only be used to create new Laravel 4 installations; but also as a means of adding packages to existing Laravel 4 applications.

Laravel 4 packages are essentially the same things as normal Composer libraries. They often utilise functionality built into the Laravel 4 framework, but this isn't a requirement of packages in general.

It follows that the end result of our work today is a stand-alone Composer library including the various deployment tools we created last time. For now, we will be placing the package code inside our application folder to speed up iteration time.

Dependency Injection

One of the darlings of modern PHP development is Dependency Injection. PHP developers have recently grown fond of unit testing and class decoupling. I'll explain both of these things shortly;

but it's important to know that they are greatly improved by the use of dependency injection.

Let's look at some non-injected dependencies, and the troubles they create for us.

```
<?php

class Archiver
{
    protected $database;

    public function __construct()
    {
        $this->database = Database::connect(
            "host",
            "username",
            "password",
            "schema"
        );
    }

    public function archive()
    {
        $entries = $this->database->getEntries();

        foreach ($entries as $entry)
        {
            if ($entry->aggregated)
            {
                $entry->archive();
            }
        }
    }
}

$archiver = new Archiver();
$archiver->archive();
```

Assuming, for a moment, that **Database** has been defined to pull all entries in such a way that they have an aggregated property and an **archive()** method; this code should be straightforward to follow.

The **Archiver** constructor initialises a database connection and stores a reference to it within a protected property. The **archive()** method iterates over the entries and archives them if they are already aggregated. Easy stuff.

Not so easy to test. The problem is that testing the business logic means testing the database connection and the entry instances as well. They are dependencies to any unit test for this class.

Furthermore; the Archiver class needs to know that these things exist and how they work. It's not enough just to know that the database instance is available or that the **getEntries()** method returns entries. If we have thirty classes all depending on the database; something small (like a change to the database password) becomes a nightmare.

Dependency injection takes two steps, in PHP. The first is to declare dependencies outside of classes and pass them in:

```
<?php

class Archiver
{
    protected $database;

    public function __construct(Database $database)
    {
        $this->database = $database;
    }

    // ...then the archive() method
}

$database = Database::connect(
    "host",
    "username",
    "password",
    "schema"
);

$archiver = new Archiver($database);
$archiver->archive();
```

This may seem like a small, tedious change but it immediately enables independent unit testing of the database and the **Archiver** class.

The second step to proper dependency injection is to abstract the dependencies in such a way that they provide a minimum of functionality. Another way of looking at it is that we want to reduce the impact of changes or the leaking of details to classes with dependencies:

```php
<?php

interface EntryProviderInterface
{
    public function getEntries();
}

class EntryProvider implements EntryProviderInterface
{
    protected $database;

    public function __construct(Database $database)
    {
        $this->database = $database;
    }

    public function getEntries()
    {
        // ...get the entries from the database
    }
}

class Archiver
{
    protected $provider;

    public function __construct(
        EntryProviderInterface $provider
    )
    {
        $this->provider = $provider;
    }

    // ...then the archive() method
}

$database = Database::connect(
    "host",
    "username",
    "password",
    "schema"
);
```

```
43
44 $provider = new EntryProvider($database);
45 $archiver = new Archiver($provider);
46 $archiver->archive();
```

Woah Nelly! That's a lot of code when compared to the first snippet; but it's worth it. Firstly; we're exposing so few details to the **Archiver** class that the entire entry dependency could be replaced in a heartbeat. This is possible because we've moved the database dependency out of the **Archiver** and into the **EntryProvider**.

We're also type-hinting an interface instead of a concrete class; which lets us swap the concrete class out with anything else that implements EntryProviderInterface. The concrete class can fetch the entries from the database, or the filesystem or whatever.

We can test the **EntryProvider** class by swapping in a fake **Database** instance. We can test the **Archiver** class by swapping in a fake **EntryProvider** instance.

So, to recap the requirements for dependency injection:

1. Don't create class instances in other classes (if they are dependencies)—pass them into the class from outside.
2. Don't type-hint concrete classes—create interfaces.

> "When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.
>
> What you should be doing is stating a need, 'I need something to drink with lunch,' and then we will make sure you have something when you sit down to eat."
>
> —John Munsch

You can learn more about Dependency Injection, from Taylor's book, at: **https://leanpub.com/laravel**.

Inversion Of Control

Inversion of Control (IoC) is the name given to the process that involves assembling class instances (and the resolution of class instances in general) within a container or registry. Where the registry

pattern involves defining class instances and then storing them in some global container; IoC involves telling the container how and where to find the instances so that they can be resolved as required.

This naturally aids dependency injection as class instances adhering to abstracted requirements can easily be resolved without first creating. To put it another way; if our classes adhere to certain requirements (via interfaces) and the IoC container can resolve them to class instances, then we don't have to do it beforehand.

The easiest way to understand this is to look at some code:

```
<?php

interface DatabaseInterface
{
    // ...
}

class Database implements DatabaseInterface
{
    // ...
}

interface EntryProviderInterface
{
    // ...
}

class EntryProvider implements EntryProviderInterface
{
    protected $database;

    public function __construct(DatabaseInterface $database)
    {
        $this->database = $database;
    }

    // ...
}

interface ArchiverInterface
{
    // ...
}
```

```
34
35 class Archiver implements ArchiverInterface
36 {
37     protected $provider;
38
39     public function __construct(
40         EntryProviderInterface $provider
41     )
42     {
43         $this->provider = $provider;
44     }
45
46     // ...
47 }
48
49 $database = new Database();
50 $provider = new EntryProvider($database);
51 $archiver = new Archiver($provider);
```

This shallowly represents a dependency (injection) chain. The last three lines are where the problem starts to become clear; the more we abstract our dependencies, the more "bootstrapping" code needs to be done every time we need the **Archiver** class.

We could abstract this by using Laravel 4's IoC container:

```
1  <?php
2
3  // ...define classes
4
5  App::bind("DatabaseInterface", function() {
6      return new Database();
7  });
8
9  App::bind("EntryProviderInterface", function() {
10     return new EntryProvider(
11         App::make("DatabaseInterface")
12     );
13 });
14
15 App::bind("ArchiverInterface", function() {
16     return new Archiver(
17         App::make("EntryProviderInterface")
18     );
```

```
19 });
20
21 $archiver = App::make("ArchiverInterface");
```

These extra nine lines (using the **App::bind()** and **App::make()** methods) tell Laravel 4 how to find/make new class instances, so we can get to the business of using them!

> You can learn more about IoC container at: **http://laravel.com/docs/ioc**.

Service Providers

The main purpose of services providers is to collect and organise the bootstrapping requirements of your package. They're not strictly a requirement of package development; but rather a Really Good Idea™.

There are three big things to service providers. The first is that they are registered in a common configuration array (in **app/config/app.php**):

```
1 'providers' => array(
2     'Illuminate\Foundation\Providers\ArtisanServiceProvider',
3     'Illuminate\Auth\AuthServiceProvider',
4     'Illuminate\Cache\CacheServiceProvider',
5     // ...
6     'Illuminate\Validation\ValidationServiceProvider',
7     'Illuminate\View\ViewServiceProvider',
8     'Illuminate\Workbench\WorkbenchServiceProvider',
9 ),
```

> This was extracted from **app/config/app.php**.

You can also add your own service providers to this list; as many packages will recommend you do. There there's the **register()** method:

```php
<?php namespace Illuminate\Cookie;

use Illuminate\Support\ServiceProvider;

class CookieServiceProvider extends ServiceProvider {

    /**
    * Register the service provider.
    *
    * @return void
    */
    public function register()
    {
        $this->app['cookie'] = $this->app->share(function($app)
        {
            $cookies = new CookieJar(
                $app['request'],
                $app['encrypter']
            );

            $config = $app['config']['session'];

            return $cookies->setDefaultPathAndDomain(
                $config['path'],
                $config['domain']
            );
        });
    }

}
```

This file should be saved as **vendor/laravel/framework/src/Illuminate/Cookie/CookieServiceProvider.php**.

When we take a look at the **register()** method of the **CookieServiceProvider** class; we can see a call to **$this->app->share()** which is similar to the **App::bind()** method but instead of creating a new class instance every time it's resolved in the IoC container; **share()** wraps a callback so that it's shared with every resolution.

The name of the **register()** method explains exactly what it should be used for; registering things

with the IoC container (which **App** extends). If you need to do other bootstrapping stuff then the method you need to use is **boot()**:

```
1  public function boot()
2  {
3      Model::setConnectionResolver($this->app['db']);
4      Model::setEventDispatcher($this->app['events']);
5  }
```

> This was extracted from **vendor/laravel/framework/src/Illuminate/Database/DatabaseServiceProvider.php**.

This **boot()** method sets two properties on the **Model** class. The same class also has a register method but these two settings are pet for the boot method.

> You can learn more about service providers at: **http://laravel.com/docs/packages#service-providers**.

Organising Code

Now that we have some tools to help us create packages; we need to port our code over from being application code to being a package. Laravel 4 provides a useful method for creating some structure to our package:

```
1  php artisan workbench Formativ/Deployment
```

You'll notice a new folder has been created, called workbench. Within this folder you'll find an assortment of files arranged in a similar way to those in the **vendor/laravel/framework/src/Illuminate/*** folders.

We need to break all of the business logic (regarding deployment) into individual classes, making use of the IoC container and dependency injection, and make a public API.

> We're not going to spend a lot of time discussing the intricacies of the deployment classes/scripts we made in the last tutorial. Refer to it if you want more of those details.

To recap; the commands we need to abstract are:

- asset:combine
- asset:minify
- clean
- copy
- distribute
- environment:get
- environment:remove
- environment:set
- watch

Many of these were cluttering up the list of commands which ship with Laravel. Our organisation should collect all of these in a common "namespace".

The first step is to create a few contracts (interfaces) for the API we want to expose through our package:

```
1  <?php
2
3  namespace Formativ\Deployment\Asset;
4
5  interface ManagerInterface
6  {
7      public function add($source, $target);
8      public function remove($target);
9      public function combine();
10     public function minify();
11 }
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/ManagerInterface.php**.

```php
<?php

namespace Formativ\Deployment\Asset;

interface WatcherInterface
{
    public function watch();
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/WatcherInterface.php**.

```php
<?php

namespace Formativ\Deployment;

interface DistributionInterface
{
    public function prepare($source, $target);
    public function sync();
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/DistributionInterface.php**.

```php
<?php

namespace Formativ\Deployment;

interface MachineInterface
{
    public function getEnvironment();
    public function setEnvironment($environment);
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/MachineInterface.php**.

The methods required by these interfaces encompass the functionality our previous commands provided for us. The next step is to fulfil these contracts with concrete class implementations:

```php
<?php

namespace Formativ\Deployment\Asset;

use Formativ\Deployment\MachineInterface;
use Illuminate\Filesystem\Filesystem;

class Manager
implements ManagerInterface
{
    protected $files;

    protected $machine;

    protected $assets = [];

    public function __construct(
        Filesystem $files,
        MachineInterface $machine
    )
    {
        $this->files = $files;
        $this->machine = $machine;
```

```
    }

    public function add($source, $target)
    {
        // ...add files to $assets
    }

    public function remove($target)
    {
        // ...remove files from $assets
    }

    public function combine()
    {
        // ...combine assets
    }

    public function minify()
    {
        // ...minify assets
    }
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/Manager.php**.

```
<?php

namespace Formativ\Deployment\Asset;

use Formativ\Deployment\MachineInterface;
use Illuminate\Filesystem\Filesystem;

class Watcher
implements WatcherInterface
{
    protected $files;

```

```
    protected $machine;

    public function __construct(
        Filesystem $files,
        MachineInterface  $machine
    )
    {
        $this->files = $files;
        $this->machine = $machine;
    }

    public function watch()
    {
        // ...watch assets for changes
    }
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/Watcher.php**.

```
<?php

namespace Formativ\Deployment;

use Formativ\Deployment\MachineInterface;
use Illuminate\Filesystem\Filesystem;

class Distribution
implements DistributionInterface
{
    protected $files;

    protected $machine;

    public function __construct(
        Filesystem $files,
        MachineInterface $machine
    )
```

```
    {
        $this->files = $files;
        $this->machine = $machine;
    }

    public function prepare($source, $target)
    {
        // ...copy + clean files for distribution
    }

    public function sync()
    {
        // ...sync distribution files to remote server
    }
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Distribution.php**.

```
<?php

namespace Formativ\Deployment;

use Illuminate\Filesystem\Filesystem;

class Machine
implements MachineInterface
{
    protected $environment;

    protected $files;

    public function __construct(Filesystem $files)
    {
        $this->files = $files;
    }

    public function getEnvironment()
```

```
20      {
21          // ...get the current environment
22      }
23
24      public function setEnvironment($environment)
25      {
26          // ...set the current environment
27      }
28  }
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Machine.php**.

The main to note about these implementations is that they use dependency injection instead of creating class instances in their constructors. As I pointed out before; we're not going to go over the actually implementation code, but feel free to port it over from the application-based commands.

As you can probably tell; I've combined related functionality into four main classes. This becomes even more apparent when you consider what the service provider has become:

```
1   <?php
2
3   namespace Formativ\Deployment;
4
5   use App;
6   use Illuminate\Support\ServiceProvider;
7
8   class DeploymentServiceProvider
9   extends ServiceProvider
10  {
11      protected $defer = true;
12
13      public function register()
14      {
15          App::bind("deployment.asset.manager", function()
16          {
17              return new Asset\Manager(
18                  App::make("files"),
19                  App::make("deployment.machine")
```

```
            );
        });

        App::bind("deployment.asset.watcher", function()
        {
            return new Asset\Watcher(
                App::make("files"),
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.distribution", function()
        {
            return new Distribution(
                App::make("files"),
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.machine", function()
        {
            return new Machine(
                App::make("files")
            );
        });

        App::bind("deployment.command.asset.combine", function()
        {
            return new Command\Asset\Combine(
                App::make("deployment.asset.manager")
            );
        });

        App::bind("deployment.command.asset.minify", function()
        {
            return new Command\Asset\Minify(
                App::make("deployment.asset.manager")
            );
        });

        App::bind("deployment.command.asset.watch", function()
        {
```

```
            return new Command\Asset\Watch(
                App::make("deployment.asset.manager")
            );
        });

        App::bind("deployment.command.distribute.prepare", function()
        {
            return new Command\Distribute\Prepare(
                App::make("deployment.distribution")
            );
        });

        App::bind("deployment.command.distribute.sync", function()
        {
            return new Command\Distribute\Sync(
                App::make("deployment.distribution")
            );
        });

        App::bind("deployment.command.machine.add", function()
        {
            return new Command\Machine\Add(
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.command.machine.remove", function()
        {
            return new Command\Machine\Remove(
                App::make("deployment.machine")
            );
        });

        $this->commands(
            "deployment.command.asset.combine",
            "deployment.command.asset.minify",
            "deployment.command.asset.watch",
            "deployment.command.distribute.prepare",
            "deployment.command.distribute.sync",
            "deployment.command.machine.add",
            "deployment.command.machine.remove"
        );
```

```
104     }
105
106     public function boot()
107     {
108         $this->package("formativ/deployment");
109         include __DIR__ . "/../../helpers.php";
110         include __DIR__ . "/../../macros.php";
111     }
112
113     public function provides()
114     {
115         return [
116             "deployment.asset.manager",
117             "deployment.asset.watcher",
118             "deployment.distribution",
119             "deployment.machine",
120             "deployment.command.asset.combine",
121             "deployment.command.asset.minify",
122             "deployment.command.asset.watch",
123             "deployment.command.distribute.prepare",
124             "deployment.command.distribute.sync",
125             "deployment.command.machine.add",
126             "deployment.command.machine.remove"
127        ];
128     }
129 }
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/DeploymentServiceProvider.php**.

I've folded the **copy()** and **clean()** methods into a single **prepare()** method.

The first four **bind()** calls bind our four main classes to the IoC container. The remaining **bind()** methods bind the artisan commands we still have to create (to replace those we make last time).

There's also a call to **$this->commands()**; which registers commands (bound to the IoC container) with artisan. Finally; we define the **provides()** method, which coincides with the **$defer = true** property, to inform Laravel which IoC container bindings are returned by this service provider. By setting **$defer = true**; we're instructing Laravel to not immediately load the provided classes and commands, but rather wait until they are required.

This is the first time we're using the **boot()** method. The call to **$this->package()** is so that we can have views, language files and other resources mapped to our package. We would be able to call them by prefixing the view names, language keys etc. with the name of the package. We're not going to be using that this time round; but it's important to know how.

We also include **helpers.php** and **macros.php** in the boot method.

```
1  <?php
2
3  namespace Formativ\Deployment\Command\Asset;
4
5  use Formativ\Deployment\Asset\ManagerInterface;
6  use Illuminate\Console\Command;
7  use Symfony\Component\Console\Input\InputArgument;
8  use Symfony\Component\Console\Input\InputOption;
9
10 class Combine
11 extends Command
12 {
13     protected $name = "deployment:asset:combine";
14
15     protected $description = "Combines multiple resource files.";
16
17     protected $manager;
18
19     public function __construct(ManagerInterface $manager)
20     {
21         parent::__construct();
22         $this->manager = $manager;
23     }
24
25     // ...
26 }
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Asset/Combine.php**.

```php
<?php

namespace Formativ\Deployment\Command\Asset;

use Formativ\Deployment\Asset\ManagerInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Minify
extends Command
{
    protected $name = "deployment:asset:minify";

    protected $description = "Minifies multiple resource files.";

    protected $manager;

    public function __construct(ManagerInterface $manager)
    {
        parent::__construct();
        $this->manager = $manager;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Asset/Minify.php**.

```php
<?php

namespace Formativ\Deployment\Command\Asset;

use Formativ\Deployment\Asset\ManagerInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Watch
extends Command
{
    protected $name = "deployment:asset:watch";

    protected $description = "Watches files for changes.";

    protected $manager;

    public function __construct(ManagerInterface $manager)
    {
       parent::__construct();
       $this->manager = $manager;
    }

    // ...
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Asset/Watch.php**.

```php
<?php

namespace Formativ\Deployment\Command\Distribute;

use Formativ\Deployment\DistributionInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Prepare
extends Command
{
    protected $name = "deployment:distribute:prepare";

    protected $description = "Prepares the distribution folder.";

    protected $distribution;

    public function __construct(
        DistributionInterface $distribution
    )
    {
        parent::__construct();
        $this->distribution = $distribution;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Distribute/Prepare.php**.

```php
<?php

namespace Formativ\Deployment\Command\Distribute;

use Formativ\Deployment\DistributionInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Sync
extends Command
{
    protected $name = "deployment:distribute:sync";

    protected $description = "Syncs changes to a target.";

    protected $distribution;

    public function __construct(
        DistributionInterface $distribution
    )
    {
        parent::__construct();
        $this->distribution = $distribution;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Distribute/Sync.php**.

```php
<?php

namespace Formativ\Deployment\Command\Machine;

use Formativ\Deployment\MachineInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Add
extends Command
{
    protected $name = "deployment:machine:add";

    protected $description = "Adds the current machine to an environment.";

    protected $machine;

    public function __construct(MachineInterface $machine)
    {
        parent::__construct();
        $this->machine = $machine;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Machine/Add.php**.

```php
<?php

namespace Formativ\Deployment\Command\Machine;

use Formativ\Deployment\MachineInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Remove
extends Command
{
    protected $name = "deployment:machine:remove";

    protected $description = "Removes the current machine from an environment.";

    protected $machine;

    public function __construct(MachineInterface $machine)
    {
        parent::__construct();
        $this->machine = $machine;
    }

    // ...
}
```

> This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Machine/Remove.php**.

> I've also omitted the **fire()** method from the new commands; consider it an exercise to add these in yourself.

Now, when we run the artisan list command; we should see all of our new package commands neatly grouped. Feel free to remove the old commands, after you've ported their functionality over to the new ones.

Publishing Configuration Files

Often you'll want to add new configuration files to the collection which ship with new Laravel applications. There's an artisan command to help with this:

```
php artisan config:publish formativ/deployment --path=workbench/Formativ/Deployme\
nt/src/config
```

This should copy the package configuration files into the **app/config/packages/formativ/deployment** directory.

> Since we have access to a whole new kind of configuration; we no longer need to override the configuration files for things like environment. As long as our helpers/macros use the package configuration files instead of the the default ones; we can leave the underlying Laravel 4 application structure (and bootstrapping code) untouched.

Creating Composer.json

Before we can publish our package, so that others can use it in their applications; we need to add a few things to the **composer.json** file that the workbench command created for us:

```
{
    "name"        : "formativ/deployment",
    "description" : "All sorts of cool things with deployment.",
    "authors"     : [
        {
            "name"  : "Christopher Pitt",
            "email" : "cgpitt@gmail.com"
        }
    ],
    "require" : {
        "php"                : ">=5.4.0",
        "illuminate/support" : "4.0.x"
    },
    "autoload" : {
        "psr-0" : {
            "Formativ\\Deployment" : "src/"
```

```
17          }
18      },
19      "minimum-stability" : "dev"
20  }
```

> This file should be saved as **workbench/formativ/deployment/composer.json**.

I've added a description, my name and email address. I also cleaned up the formatting a bit; but that's not really a requirement. I've also set the minimum PHP version to 5.4.0 as we're using things like short array syntax in our package.

> You should also create a **readme.md** file, which contains installation instructions, usage instructions, license and contribution information. I can't understate the import ants of these things—they are your only hope to attract contributors.

> You can learn more about the composer.json file at: **http://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup**.

Submitting A Package To Packagist

To get others using your packages; you need to submit them to packagist.org. Go to **http://packagist.org** and sign in. If you haven't already got an account, you should create it and link to your GitHub account.

You'll find a big "Submit Package" button on the home page. Click it and paste the URL to your GitHub repository in the text field. Submit the form and you should be good to go.

> You can learn more about package development at: **http://laravel.com/docs/packages**.

Note On Testing

Those of you who already know about unit testing will doubtlessly wonder where the section on testing is. I ran short on time for that in this chapter, but it will be the subject of a future chapter; which will demonstrate the benefits of dependency injection as it relates to unit testing.

Real Time Chat

One of the most pervasive and least understood technologies that underpin the internet is socket programming. It's been a dark art for decades, but the recent standardisation of web sockets (in relation to HTML 5) has made this type of programming a little easier to get into.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-real-time-chat**.

Dependencies

This tutorial depends heavily on client-side resources. We're developing a Laravel 4 application which has lots of server-side aspects; but it's also a chat app. There be scripts!

Bootstrap

For this; we're using Bootstrap and EmberJS. Download Bootstrap at: **http://getbootstrap.com/** and unpack it into your public folder. Where you put the individual files makes little difference, but I have put the scripts in **public/js**, the stylesheets in **public/css** and the fonts in **public/fonts**. Where you see those paths in my source-code; you should substitute them with your own.

EmberJS

Next up, download EmberJS at: **http://emberjs.com/** and unpack it into your public folder. You'll also need the Ember.Data script at: **http://emberjs.com/guides/getting-started/obtaining-emberjs-and-dependencies/**.

Ratchet

For the server-side portion of dependencies, we need to download a library called Ratchet. I'll explain it shortly, but in the meantime we need to add it to our **composer.json** file:

```
"require" : {
    "laravel/framework" : "4.0.*",
    "cboden/Ratchet"    : "0.3.*"
},
```

> This was extracted from **composer.json**.

Follow that up with:

```
composer update
```

> Ratchet isn't built specifically for Laravel 4, so there are no service providers for us to add.

We'll now have access to the Ratchet library for client-server communication, Bootstrap for styling the interface and EmberJS for connecting these two things together.

ReactPHP

Before we can understand Ratchet, we need to understand ReactPHP. ReactPHP was born out of the need to develop event-based, asynchronous PHP applications. If you've worked with Node.JS you'll feel right at home developing applications with ReactPHP; as they share a similar approaches to code. We're not going to develop our chat application in ReactPHP, but it's a dependency for Ratchet...

> You can learn more about ReactPHP at: **http://reactphp.org/**.

Ratchet

One of the many ways in which real-time client-server applications are made possible is by what's called socket programming. Believe it or not; most of what you do on the internet depends on socket

programming. From simple browsing to streaming—your computer opens a socket connection to a server and the server sends data back through it.

PHP supports this type of programming but PHP websites have not typically been developed with this kind of model in mind. PHP developers have preferred the typical request/response model, and it's comparatively easier than low-level socket programming.

Enter ReactPHP. One of the requirements for building a fully-capable socket programming framework is creating what's called an Event Loop. ReactPHP has this and Ratchet uses it, along with the Publish/Subscribe model to accept and maintain open socket connections.

ReactPHP wraps the low-level PHP functions into a nice socket programming API and Ratchet wraps that API into another API that's even easier to use.

> You can learn more about Ratchet at: **http://socketo.me/**.

Creating An Interface

Let's get to the code! We're going to need an interface (kind of like a wireframe) so we know what to build with our application. Let's set up a simple view and plug it into EmberJS.

> I should mention that I am by no means an EmberJS expert. I learned all I know of it, while writing this tutorial, by following various guides. The point of this is not to teach EmberJS so much as it is to show EmberJS integration with Laravel 4.

Creating A View

Let's change the default routes.php file to load a custom view:

```
1  <?php
2
3  Route::get("/", function()
4  {
5      return View::make("index/index");
6  });
```

This file should be saved as **app/routes.php**.

Then we need to add this view:

```
<!DOCTYPE html>
<html lang="en">
    <head>
    <meta charset="UTF-8" />
        <link
            rel="stylesheet"
            type="text/css"
            href="{{ asset("css/bootstrap.3.0.0.css") }}"
        />
        <link
            rel="stylesheet"
            type="text/css"
            href="{{ asset("css/bootstrap.theme.3.0.0.css") }}"
        />
        <title>Laravel 4 Chat</title>
    </head>
    <body>
        <script type="text/x-handlebars">
            @{{outlet}}
        </script>
        <script
            type="text/x-handlebars"
            data-template-name="index"
        >
            <div class="container">
                <div class="row">
                    <div class="col-md-12">
                        <h1>Laravel 4 Chat</h1>
                        <table class="table table-striped">
                            @{{#each}}
                                <tr>
                                    <td>
                                        @{{user}}
                                    </td>
                                    <td>
```

```
                                    @{{text}}
                                </td>
                            </tr>
                        @{{/each}}
                    </table>
                </div>
            </div>
            <div class="row">
                <div class="col-md-12">
                    <div class="input-group">
                        <input
                            type="text"
                            class="form-control"
                        />
                        <span class="input-group-btn">
                            <button class="btn btn-default">
                                Send
                            </button>
                        </span>
                    </div>
                </div>
            </div>
        </div>
    </script>
    <script
        type="text/javascript"
        src="{{ asset("js/jquery.1.9.1.js") }}"
    ></script>
    <script
        type="text/javascript"
        src="{{ asset("js/handlebars.1.0.0.js") }}"
    ></script>
    <script
        type="text/javascript"
        src="{{ asset("js/ember.1.1.1.js") }}"
    ></script>
    <script
        type="text/javascript"
        src="{{ asset("js/ember.data.1.0.0.js") }}"
    ></script>
    <script
        type="text/javascript"
```

```
78              src="{{ asset("js/bootstrap.3.0.0.js") }}"
79          ></script>
80          <script
81              type="text/javascript"
82              src="{{ asset("js/shared.js") }}"
83          ></script>
84      </body>
85  </html>
```

This file should be saved as **app/views/index/index.blade.php**.

Both Blade and EmberJS use double-curly-brace syntax for variable and logic substitution. Luckily Blade includes a mechanism to ignore curly brace blocks, by prepending them with @ symbols. Thus our template includes @ symbols before all EmberJS blocks.

The scripts and stylesheets need to be relative to where you saved them or you're going to see errors.

Creating An EmberJS App

You'll notice I've specified **shared.css** and **shared.js** files—the CSS file is blank, but the JavaScript file contains:

```
// 1
var App = Ember.Application.create();

// 2
App.Router.map(function() {
    this.resource("index", {
        "path" : "/"
    });
});

// 3
App.Message = DS.Model.extend({
    "user" : DS.attr("string"),
    "text" : DS.attr("string")
});

// 4
App.ApplicationAdapter = DS.FixtureAdapter.extend();

// 5
App.Message.FIXTURES = [
    {
        "id"   : 1,
        "user" : "Chris",
        "text" : "Hello World."
    },
    {
        "id"   : 2,
        "user" : "Wayne",
        "text" : "Don't dig it, man."
    },
    {
        "id"   : 3,
        "user" : "Chris",
        "text" : "Meh."
    }
];
```

This file should be saved as **public/js/shared.js**.

If you're an EmberJS noob, like me, then it will help to understand what each piece of this script is doing.

1. We create a new Ember application with **Ember.Application.create()**.
2. Routes are defined in the **App.Route.map()** method, and we tell the application to equate the path / to the index resource.
3. We define a **Message** model. These are similar to the Eloquent models we have in Laravel 4, but they're built to work with EmberJS (and are obviously on the client-side of the application).
4. We specify a fixture-based data store for our application. We're using this, temporarily, to fill our interface with some dummy data, but we'll add a dynamic data store before too long…
5. Here we add the fixture data. Notice that, in addition to the two model fields we defined, we also specify ID values for the fixture rows. This data is used to single out individual Message objects.

When you browse to the base URL of the application; you should now see an acceptably styled list of message objects, along with a heading and input form. Let's make it dynamic!

You should also see some console log messages (depending on your browser) which show that EmberJS is running.

Creating A Service Provider

Following on from a previous tutorial; we're going to be creating a service provider which will provide the various classes for our application. Create a new workbench:

```
1  php artisan workbench formativ/chat
```

This will produce (amongst other things) a service provider template. I've added a few IoC bindings to it:

```
1  <?php
2
3  namespace Formativ\Chat;
4
5  use Evenement\EventEmitter;
6  use Illuminate\Support\ServiceProvider;
7  use Ratchet\Server\IoServer;
8
9  class ChatServiceProvider
10 extends ServiceProvider
11 {
```

```
12      protected $defer = true;
13
14      public function register()
15      {
16          $this->app->bind("chat.emitter", function()
17          {
18              return new EventEmitter();
19          });
20
21          $this->app->bind("chat.chat", function()
22          {
23              return new Chat(
24                  $this->app->make("chat.emitter")
25              );
26          });
27
28          $this->app->bind("chat.user", function()
29          {
30              return new User();
31          });
32
33          $this->app->bind("chat.command.serve", function()
34          {
35              return new Command\Serve(
36                  $this->app->make("chat.chat")
37              );
38          });
39
40          $this->commands("chat.command.serve");
41      }
42
43      public function provides()
44      {
45          return [
46              "chat.chat",
47              "chat.command.serve",
48              "chat.emitter",
49              "chat.server"
50          ];
51      }
52  }
```

> This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/ChatServiceProvider.php**.

The first binding is a simple alias to the **Evenement\EventEmitter** class (which Ratchet requires). We bind it here as we cannot guarantee that Ratchet will continue to use **Evenement\EventEmitter** and we'll need a reliable way to unit test possible alternatives in the future.

Creating A Chat Handler

Let's look closer at the second and third bindings. The first is to the **Chat** class. It implements the **ChatInterface** interface:

```
1  <?php
2
3  namespace Formativ\Chat;
4
5  use Evenement\EventEmitterInterface;
6  use Ratchet\ConnectionInterface;
7  use Ratchet\MessageComponentInterface;
8
9  interface ChatInterface
10 extends MessageComponentInterface
11 {
12     public function getUserBySocket(ConnectionInterface $socket);
13     public function getEmitter();
14     public function setEmitter(EventEmitterInterface $emitter);
15     public function getUsers();
16 }
```

> This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/ChatInterface.php**.

It's interesting to note that PHP actually supports interfaces which extend other interfaces. This is useful if you want to expect a certain level of functionality, provided by a third-party library (such as SPL), but want to add your own requirements on top.

The concrete implementation looks like this:

```php
<?php

namespace Formativ\Chat;

use Evenement\EventEmitterInterface;
use Exception;
use Ratchet\ConnectionInterface;
use SplObjectStorage;

class Chat
implements ChatInterface
{
    protected $users;
    protected $emitter;
    protected $id = 1;

    public function getUserBySocket(ConnectionInterface $socket)
    {
        foreach ($this->users as $next)
        {
            if ($next->getSocket() === $socket)
            {
                return $next;
            }
        }

        return null;
    }

    public function getEmitter()
    {
        return $this->emitter;
    }

    public function setEmitter(EventEmitterInterface $emitter)
    {
        $this->emitter = $emitter;
    }

    public function getUsers()
    {
        return $this->users;
```

```
43      }
44
45      public function __construct(EventEmitterInterface $emitter)
46      {
47          $this->emitter = $emitter;
48          $this->users   = new SplObjectStorage();
49      }
50
51      public function onOpen(ConnectionInterface $socket)
52      {
53          $user = new User();
54          $user->setId($this->id++);
55          $user->setSocket($socket);
56
57          $this->users->attach($user);
58          $this->emitter->emit("open", [$user]);
59      }
60
61      public function onMessage(
62          ConnectionInterface $socket,
63          $message
64      )
65      {
66          $user = $this->getUserBySocket($socket);
67          $message = json_decode($message);
68
69          switch ($message->type)
70          {
71              case "name":
72              {
73                  $user->setName($message->data);
74                  $this->emitter->emit("name", [
75                      $user,
76                      $message->data
77                  ]);
78                  break;
79              }
80
81              case "message":
82              {
83                  $this->emitter->emit("message", [
84                      $user,
```

```
                    $message->data
                ]);
                break;
            }
        }

        foreach ($this->users as $next)
        {
            if ($next !== $user)
            {
                $next->getSocket()->send(json_encode([
                    "user" => [
                        "id"   => $user->getId(),
                        "name" => $user->getName()
                    ],
                    "message" => $message
                ]));
            }
        }
    }

    public function onClose(ConnectionInterface $socket)
    {
        $user = $this->getUserBySocket($socket);

        if ($user)
        {
            $this->users->detach($user);
            $this->emitter->emit("close", [$user]);
        }
    }

    public function onError(
        ConnectionInterface $socket,
        Exception $exception
    )
    {
        $user = $this->getUserBySocket($socket);

        if ($user)
        {
            $user->getSocket()->close();
```

```
127             $this->emitter->emit("error", [$user, $exception]);
128         }
129     }
130 }
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/Chat.php**.

It's fairly simple: the (delegate) **onOpen** and **onClose** methods handle creating new User objects and disposing of them. The **onMessage** method translates JSON-encoded message objects into required actions and responds back to the other socket connections with further details.

Creating A Socket Wrapper

Additionally; the **UserInterface** interface and **User** class look like this:

```
1  <?php
2
3  namespace Formativ\Chat;
4
5  use Evenement\EventEmitterInterface;
6  use Ratchet\ConnectionInterface;
7  use Ratchet\MessageComponentInterface;
8
9  interface UserInterface
10 {
11     public function getSocket();
12     public function setSocket(ConnectionInterface $socket);
13     public function getId();
14     public function setId($id);
15     public function getName();
16     public function setName($name);
17 }
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/UserInterface.php**.

```php
<?php

namespace Formativ\Chat;

use Ratchet\ConnectionInterface;

class User
implements UserInterface
{
    protected $socket;
    protected $id;
    protected $name;

    public function getSocket()
    {
        return $this->socket;
    }

    public function setSocket(ConnectionInterface $socket)
    {
        $this->socket = $socket;
        return $this;
    }

    public function getId()
    {
        return $this->id;
    }

    public function setId($id)
    {
        $this->id = $id;
        return $this;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
```

```
43          $this->name = $name;
44          return $this;
45      }
46  }
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/User.php**.

The **User** class is a simple wrapper for a socket resource and name string. The way we've chosen to implement the Ratchet server requires that we have a class which implements the **MessageComponentInterface** interface; and this interface specifies that **ConnectionInterface** objects are passed back and forth. There's no way to identify these, by name (and id), so we're adding that functionality with the extra layer.

Creating A Serve Command

All these classes lead us to the artisan command which will kick things off:

```
1   <?php
2
3   namespace Formativ\Chat\Command;
4
5   use Illuminate\Console\Command;
6   use Formativ\Chat\ChatInterface;
7   use Formativ\Chat\UserInterface;
8   use Ratchet\ConnectionInterface;
9   use Ratchet\Http\HttpServer;
10  use Ratchet\Server\IoServer;
11  use Ratchet\WebSocket\WsServer;
12  use Symfony\Component\Console\Input\InputOption;
13  use Symfony\Component\Console\Input\InputArgument;
14
15  class Serve
16  extends Command
17  {
18      protected $name        = "chat:serve";
19      protected $description = "Command description.";
20      protected $chat;
21
```

```
22      protected function getUserName($user)
23      {
24          $suffix = " (" . $user->getId() . ")";
25
26          if ($name = $user->getName())
27          {
28              return $name . $suffix;
29          }
30
31          return "User" . $suffix;
32      }
33
34      public function __construct(ChatInterface $chat)
35      {
36          parent::__construct();
37
38          $this->chat = $chat;
39
40          $open = function(UserInterface $user)
41          {
42              $name = $this->getUserName($user);
43              $this->line("
44                  <info>" . $name . " connected.</info>
45              ");
46          };
47
48          $this->chat->getEmitter()->on("open", $open);
49
50          $close = function(UserInterface $user)
51          {
52              $name = $this->getUserName($user);
53              $this->line("
54                  <info>" . $name . " disconnected.</info>
55              ");
56          };
57
58          $this->chat->getEmitter()->on("close", $close);
59
60          $message = function(UserInterface $user, $message)
61          {
62              $name = $this->getUserName($user);
63              $this->line("
```

```
64                  <info>New message from " . $name . ":</info>
65                  <comment>" . $message . "</comment>
66                  <info>.</info>
67              ");
68          };
69
70          $this->chat->getEmitter()->on("message", $message);
71
72          $name = function(UserInterface $user, $message)
73          {
74              $this->line("
75                  <info>User changed their name to:</info>
76                  <comment>" . $message . "</comment>
77                  <info>.</info>
78              ");
79          };
80
81          $this->chat->getEmitter()->on("name", $name);
82
83          $error = function(UserInterface $user, $exception)
84          {
85              $message = $exception->getMessage();
86
87              $this->line("
88                  <info>User encountered an exception:</info>
89                  <comment>" . $message . "</comment>
90                  <info>.</info>
91              ");
92          };
93
94          $this->chat->getEmitter()->on("error", $error);
95      }
96
97      public function fire()
98      {
99          $port = (integer) $this->option("port");
100
101         if (!$port)
102         {
103             $port = 7778;
104         }
105
```

```
106         $server = IoServer::factory(
107             new HttpServer(
108                 new WsServer(
109                     $this->chat
110                 )
111             ),
112             $port
113         );
114 
115         $this->line("
116             <info>Listening on port</info>
117             <comment>" . $port . "</comment>
118             <info>.</info>
119         ");
120 
121         $server->run();
122     }
123 
124     protected function getOptions()
125     {
126         return [
127             [
128                 "port",
129                 null,
130                 InputOption::VALUE_REQUIRED,
131                 "Port to listen on.",
132                 null
133             ]
134         ];
135     }
136 }
```

> This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/Command/Serve.php**.

The reason for us adding the event emitter to the equation should now be obvious—we need a way to tie into the delegated events, of the **Chat** class, without leaking the abstraction we gain from it. In other words; we don't want the **Chat** class to know of the existence of the artisan command. Similarly; we don't want the artisan command to know of the **onOpen**, **onMessage**, **onError** and

onMessage methods so instead we use a publish/subscribe model for notifying the command of changes. The result is a clean abstraction.

The **fire()** method gets (or defaults) the port and starts the Ratchet web socket server.

> The method we're using, to start the server, is not the only way it can be started. You can learn more about the web socket server at: **http://socketo.me/docs/websocket**.

Connecting To The Socket Server

To connect to the web socket server; we need to add a bit of vanilla JavaScript:

```
try {
    if (!WebSocket) {
        console.log("no websocket support");
    } else {
        var socket = new WebSocket("ws://127.0.0.1:7778/");

        socket.addEventListener("open", function (e) {
            console.log("open: ", e);
        });

        socket.addEventListener("error", function (e) {
            console.log("error: ", e);
        });

        socket.addEventListener("message", function (e) {
            console.log("message: ", JSON.parse(e.data));
        });

        console.log("socket:", socket);

        window.socket = socket;
    }
} catch (e) {
    console.log("exception: " + e);
}
```

> This was extracted from **public/js/shared.js**.

We've wrapped this code in a try-catch block as not all browsers support Web Sockets yet. There are a number of libraries which will shim this functionality, but their use is outside the scope of this tutorial.

> You can find a couple of these libraries at: **https://github.com/gimite/web-socket-js** and **https://github.com/sockjs**.

This code will attempt to open a socket connection to **127.0.0.1:7778** (the address and port used in the serve command) and write some console messages depending on the events that are emitted. You'll notice we're also assigning the socket instance to the window object; so we can send some debugging commands through it.

This allows us to see both the server-side of things, as well as the client-side...

Wiring Up The Interface

Getting our interface talking to our socket server is relatively straightforward. We begin by disabling our fixture data and modifying our model slightly:

```
App.Message = DS.Model.extend({
    "user_id"       : DS.attr("integer"),
    "user_name"     : DS.attr("string"),
    "user_id_class" : DS.attr("string"),
    "message"       : DS.attr("string")
});

App.ApplicationAdapter = DS.FixtureAdapter.extend();

App.Message.FIXTURES = [
    // {
    //     "id"   : 1,
    //     "user" : "Chris",
    //     "text" : "Hello World."
    // },
```

```
    // {
    //     "id"   : 2,
    //     "user" : "Wayne",
    //     "text" : "Don't dig it, man."
    // },
    // {
    //     "id"   : 3,
    //     "user" : "Chris",
    //     "text" : "Meh."
    // }
];
```

This was extracted from **public/js/shared.js**.

If you want to pre-populate your chat application with a history; you could feed this fixture configuration with data from your server.

Showing Chat Messages

Now the index template will show only the heading and form elements, but no chat messages. In order to populate these; we need to store a reference to the application data store:

```
var store;

App.IndexRoute = Ember.Route.extend({
    "init" : function() {
        store = this.store;
    },
    "model" : function () {
        return store.find("message");
    }
});
```

> This was extracted from **public/js/shared.js**.

We store this reference because we will need to push rows into the store once we receive them from the open socket. This leads us to the changes to web sockets:

```
try {
    var id = 1;

    if (!WebSocket) {
        console.log("no websocket support");
    } else {

        var socket = new WebSocket("ws://127.0.0.1:7778/");
        var id     = 1;

        socket.addEventListener("open", function (e) {
            // console.log("open: ", e);
        });

        socket.addEventListener("error", function (e) {
            console.log("error: ", e);
        });

        socket.addEventListener("message", function (e) {

            var data      = JSON.parse(e.data);
            var user_id   = data.user.id;
            var user_name = data.user.name;
            var message   = data.message.data;

            switch (data.message.type) {

                case "name":
                    $(".name-" + user_id).html(user_name);
                    break;

                case "message":
                    store.push("message", {
                        "id"            : id++,
```

```
                        "user_id"       : user_id,
                        "user_name"     : user_name || "User",
                        "user_id_class" : "name-" + user_id,
                        "message"       : message
                    });
                    break;

            }

        });

        // console.log("socket:", socket);

        window.socket = socket; // debug
    }
} catch (e) {
    console.log("exception: " + e);
}
```

This was extracted from **public/js/shared.js**.

We start by defining an id variable, to store the id's of message objects as they are passed through the socket. Inside the **onMessage** event handler; we parse the JSON data string and determine the type of message being received. If it's a name message (a user changing their name) then we update all the table cells matching the user's server id. If it's a normal message object; we push it into the data store.

Sending Chat Messages

This gets us part of the way, since console message commands will visually affect the UI. We still need to wire up the input form...

```
App.IndexController = Ember.ArrayController.extend({
    "command" : null,
    "actions" : {
        "send" : function(key) {

            if (key && key != 13) {
                return;
            }

            var command = this.get("command") || "";

            if (command.indexOf("/name") === 0) {
                socket.send(JSON.stringify({
                    "type" : "name",
                    "data" : command.split("/name")[1]
                }));
            } else {
                socket.send(JSON.stringify({
                    "type" : "message",
                    "data" : command
                }));
            }

            this.set("command", null);
        }
    }
});

App.IndexView = Ember.View.extend({
    "keyDown" : function(e) {
        this.get("controller").send("send", e.keyCode);
    }
});
```

This was extracted from **public/js/shared.js**.

We create **IndexController** and **IndexView** objects. The **IndexView** object intercepts the **keyDown** event and passes it to the **IndexController** object. The first bit of logic tells the **send()** method to ignore all keystrokes that aren't the enter key. This means enter will trigger the **send()** method.

It continues by checking for the presence of a **/name** command switch. If that's present in the input value (via **this.get("command")**) then it sends a message to the server to change the user's name. Otherwise it sends a normal message to the server. In order for the UI to update for the person sending the message; we need to also slightly modify the **Chat** class:

```
public function onMessage(ConnectionInterface $socket, $message)
{
    $user = $this->getUserBySocket($socket);
    $message = json_decode($message);

    switch ($message->type)
    {
        case "name":
        {
            $user->setName($message->data);
            $this->emitter->emit("name", [
                $user,
                $message->data
            ]);
            break;
        }

        case "message":
        {
            $this->emitter->emit("message", [
                $user,
                $message->data
            ]);
            break;
        }
    }

    foreach ($this->users as $next)
    {
        // if ($next !== $user)
        // {
            $next->getSocket()->send(json_encode([
                "user" => [
                    "id"   => $user->getId(),
                    "name" => $user->getName()
                ],
                "message" => $message
            ]));
```

```
        // }
    }
}
```

> This was extracted from **workbench/formativ/chat/src/Formativ/Chat/Chat.php**.

The change we've made is to exclude the logic which prevented messages from being sent to the user from which they came. All messages will essentially be sent to everyone on the server now.

Finishing Up The Template

The final change is to the index template, as we changed the model structure and need to adjust for this in the template:

```
<script
    type="text/x-handlebars"
    data-template-name="index"
>
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <h1>Laravel 4 Chat</h1>
                <table class="table table-striped">
                    @{{#each message in model}}
                        <tr>
                            <td @{{bind-attr class="message.user_id_class"}}>
                                @{{message.user_name}}
                            </td>
                            <td>
                                @{{message.message}}
                            </td>
                        </tr>
                    @{{/each}}
                </table>
            </div>
        </div>
        <div class="row">
            <div class="col-md-12">
```

```
25                  <div class="input-group">
26                      @{{input
27                          type="text"
28                          value=command
29                          class="form-control"
30                      }}
31                      <span class="input-group-btn">
32                          <button
33                              class="btn btn-default"
34                              @{{action "send"}}
35                          >
36                              Send
37                          </button>
38                      </span>
39                  </div>
40              </div>
41          </div>
42      </div>
43  </script>
```

> This was extracted from **app/views/index/index.blade.php**.

You'll notice, apart from us using different field names; that we've change how the loop is done, the class added to each "label" cell and how the input field is generated.

The reason for the change to the loop structure is simply so that you can see another way to represented enumerable data in handlebars. Where previously we used a simple **{{#each}}** tag, we're now being more explicit about the data we want to iterate.

We add a special class on each "label" cell as we need to target these cells and change their contents, in the event that a user decides to change their name.

Finally, we change how the input field is generated because we need to bind its value to the **IndexController**'s command property. This format allows that succinctly.

> You can learn more about Ember JS at: **http://emberjs.com/**.

Note On Nginx

For persistent sockets to remain open, Apache needs to keep a single process thread occupied. This is a problem as it will consume vast amounts of RAM over a shorter time period than non-persistent connections would. For this reason; I highly recommend using either Nginx or an event-based language to create your chat application.

It's not that Apache sucks; this is just not the sort of thing it was designed for. Nginx, on the other hand, is event-based and doesn't hold onto a whole thread while it waits for activity through the open socket.

> You can learn more about Nginx at: **http://wiki.nginx.org/Main**.

Multisites

If you've ever had to build an application that shares some business logic, and even interface logic, across multiple interfaces then you will undoubtedly have debated the merits of instead creating multiple applications or finding some way to handle each interface within the same codebase.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-multisites**.

Note on Operating Systems

I work on a Macbook, and deal with Linux servers every day. I seldom interact with Windows-based machines. As a result; most of the work I do is never run on Windows. Remember this as you follow this chapter—I will not show how to do things on Windows because it's dead to me. If you are forced to use it; then you have my sympathies.

Note on Server Setup

We're going to look at how to create virtual hosts in Apache2 and Nginx, but we won't look at how to get those systems running int he first place. It's not something I consider particularly tricky, and the internet is filled with wonderful tutorials on the subject.

Note on Dutch

I use it in this chapter, but I don't really speak it. Google Translate does. Blame Google Translate.

Virtual Hosts

It may surprise you to know that single domain names do not translate into single web servers. Modern web servers have the ability to load many different domains, and these domains are often referred to as Virtual Hosts or Virtual Domains. We're going to see how to use them with Laravel 4.

Adding Virtual Host Entries

When you type an address into your browser, and hit enter; your browser goes through a set of steps in order to get you the web page you want. First, it queries what's called a hosts file to see if the address you typed in is a reference to a local IP address. If not found, the browser then queries whatever DNS servers are available to the operating system.

A DNS server compares an address (e.g. example.com) with a list of IP addresses it has on file. If an IP address is found; the browser's request is forwarded to it and the web page is delivered.

> This is a rather simplified explanation of the steps involved. You can probably find more details at: **http://en.wikipedia.org/wiki/Internet**.

In a sense; the hosts file acts as a DNS server before any remote requests are made. It follows that the best place to add references to local IP addresses (local web servers) is in the hosts file.

To do that, open the hosts file:

```
1 sudo vim /etc/hosts
```

> If that command gives you errors then you can try running it without the **sudo** part. Sudo is (simply speaking) a way to run a command at the highest permission level possible; to ensure it executes correctly. If may be that you do not have sufficient privileges to run commands as root (with **sudo**).

> Vim is a teminal-based text editor. If you're uneasy using it; what we're after is editing **/etc/hosts**.

On a new line, at the bottom of **/etc/hosts**, add:

```
1 127.0.0.1 dev.tutorial
```

> This was extracted from **/etc/hosts**.

This line tells the operating system to send requests to dev.tutorial to **127.0.0.1**. You'll want to replace **127.0.0.1** with the IP address of the server you are using for this tutorial (assuming it's not LAMP/MAMP on your local machine) and **dev.tutorial** with whatever you want the virtual host to be.

> You'll need to repeat this process for each virtual host you add. If your application has (for example) three different domains pointing to it; then you will need to set three of these up to test them locally.

Creating Apache 2 Virtual Hosts

Apache2 virtual host entries are created by manipulating the configuration files usually located in **/etc/apache2/sites-available/**. You can edit the default file, and add the entries to it, or you can create new files.

> If you choose to create new files for your virtual host entries, then you will need to symlink them to **/etc/apache2/sites-enabled/*** where ***** matches the name of the files you create in the **sites-available** folder.

The expanded syntax of virtual host entries can be quite a bit to type/maintain; so I have a few go-to lines I usually use:

```
1 <VirtualHost *:80>
2     DocumentRoot /var/www/site1
3     ServerName dev.site1
4 </VirtualHost>
```

> This was extracted from **/etc/apache2/sites-available/default**.

The **ServerName** property can be anything you like—it's the address you will use in your browser. The **DocumentRoot** property needs to point to an existing folder in the filesystem of the web server machine.

> You can learn more about virtual host entries, in Apache2, at: **http://httpd.apache.org/docs/current/vhosts/examples.html**.

Creating Nginx Virtual Hosts

Similar to Apache2, Nginx virtual host entries are created by manipulating the configuration files usually located in **/etc/nginx/sites-available/**. You can edit the **default** file, and add the entries to it, or you can create new files.

> If you choose to create new files for your virtual host entries, then you will need to symlink them to **/etc/nginx/sites-enabled/*** where ***** matches the name of the files you create in the **sites-available** folder.

The expanded syntax of virtual host entries can be quite a bit to type/maintain; so I have a few go-to lines I usually use:

```
server {
    listen 80;
    server_name dev.site1;
    root /var/www/site1;
    index index.php;

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php
        include fastcgi_params;
    }
}
```

This was extracted from **/etc/nginx/sites-available/default**.

This site definition assumes you are using PHP5 FPM. Getting this installed and running is out of the scope of this tutorial, but there are plenty of great tutorials on the subject.

You can learn more about virtual host entries, in Nginx, at: **https://www.digitalocean.com/community/articles/how-to-set-up-nginx-virtual-hosts-server-blocks-on-ubuntu-12-04-lts–3**.

Environments

Laravel 4 employs a system of execution environments. Think of these as different contexts in which different configuration files will be loaded; determined by the host name of the machine or command flags.

To illustrate this concept; think of the differences between developing and testing your applications locally and running them on a production server. You will need to target different databases, use different caching schemes etc.

This can be achieved, deterministically, by setting the environments to match the host names of the machines the code will be executed (local and production respectively) and having different sets of configuration files for each environment.

When a Laravel 4 application is executed, it will determine the environment that you are running it in, and adjust the path to configuration files accordingly. This approach has two caveats.

The first caveat is that the PHP configuration files in **app/config** are always loaded and environment-based configuration files are then loaded on top of them. If you have a database set up in **app/config/database.php** and nothing in **app/config/production/database.php**, the global database configuration details will apply.

The second caveat is that you can override the environment Laravel 4 would otherwise use by supplying an environment flag to artisan commands:

```
php artisan migrate --env=local
```

This will tell artisan to execute the database migrations in the local environment, whether or not the environment you are running it in is local.

This is important to multisites because Laravel 4 configuration files have the ability to connect to different database, load different views and send different emails based on environmental configuration.

Note on Running Commands in Local Environment

The moment you add multiple environments to your application, you create the possibility that artisan commands might be run on the incorrect environment.

Just because Laravel 4 is capable of executing in the correct environment doesn't mean you will always remember which environment you are in or which environment you should be in…

Start learning to provide the environment for every artisan command, when you're working with multiple environments. It's a good habit to get into.

Using Site-Specific Views

One of the benefits of multiple environment-specific configuration sets is that we can load different views for different sites. Let's begin by creating a few:

```
127.0.0.1 dev.www.tutorial-laravel-4-multisites
127.0.0.1 dev.admin.tutorial-laravel-4-multisites
```

> This was extracted from **/etc/hosts**.

You can really create any domains you want, but for this part we're going to need multiple domains pointing to our testing server so that we can actually see different view files being loaded, based on domain.

Next, update your **app/bootstrap/start.php** file to include the virtual host domains you've created:

```
$env = $app->detectEnvironment([
    "www"   => ["dev.www.tutorial-laravel-4-multisites"],
    "admin" => ["dev.admin.tutorial-laravel-4-multisites"]
]);
```

> This was extracted from **/bootstrap/start.php**.

> Phil Sturgeon made an excellent point[a] about not using URL's to toggle environments, as somebody can easily set a hostname to match your development environment and point it to your production environment to cause unexpected results. This can lead to people gleaning more information than you would expect via debugging information or backtraces.
>
> [a] http://www.reddit.com/r/PHP/comments/1qm8e3/tutorial_multisites_with_laravel_4/cdecd88

If you would rather determine the current environment via a server configuration property; you can pass a callback to the **detectEnvironment()** method:

```
$env = $app->detectEnvironment(function()
{
    return Input::server("environment", "development");
});
```

> This was extracted from **/bootstrap/start.php**.

Clean out the **app/views** folder and create **www** and **admin** folders, each with their own **layout.blade.php** and **index/index.blade.php** files.

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Laravel 4 Multisites</title>
    </head>
    <body>
        @yield("content")
    </body>
</html>
```

This file should be saved as **app/views/www/layout.blade.php**.

```
@extends("layout")
@section("content")
    Welcome to our website!
@stop
```

This file should be saved as **app/views/www/index/index.blade.php**.

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Laravel 4 Multisites - Admin</title>
    </head>
    <body>
        @yield("content")
    </body>
</html>
```

This file should be saved as **app/views/admin/layout.blade.php**.

```
@extends("layout")
@section("content")
    Please log in to use the admin.
@stop
```

This file should be saved as **app/views/admin/index/index.blade.php**.

Let's also prepare the routes and controllers for the rest of the tutorial, by updating both:

```
<?php

Route::any("/", [
    "as"   => "index/index",
    "uses" => "IndexController@indexAction"
]);
```

This file should be saved as **app/routes.php**.

```
<?php

class IndexController
extends BaseController
{
    public function indexAction()
    {
        return View::make("index/index");
    }
}
```

This file should be saved as **app/controllers/IndexController.php**.

In order for Laravel to know which views to use for each environment, we should also create the configuration files for the environments.

```
<?php

return [
    "paths" => [app_path() . "/views/www"]
];
```

This file should be saved as **app/config/www/view.php**.

```
<?php

return [
    "paths" => [app_path() . "/views/admin"]
];
```

This file should be saved as **app/config/admin/view.php**.

These new configuration files tell Laravel 4 not only to look for views in the default directory but also to look within the environment-specific view folders we've set up. Going to each of these virtual host domains should now render different content.

I would recommend this approach for sites that need to drastically change their appearance in ways mere CSS couldn't achieve. If, for example, your different sites need to load different HTML or extra components then this is a good approach to take.

Another good time to use this kind of thing is when you want to separate the markup of a CMS from that of a client-facing website. Both would need access to the same business logic and storage system, but their interfaces should be vastly different.

I've also used this approach when I've needed to create basic mobile interfaces and rich interactive interfaces for the same brands...

> You can learn more about environments at: **http://laravel.com/docs/configuration#environment-configuration**.

Using Site-Specific Routes

Another aspect to multiple-domain applications is how they can affect (and be affected by) the routes file. Consider the following route groups:

```
Route::group([
    "domain" => "dev.www.tutorial-laravel-4-multisites"
], function()
{
    Route::any("/about", function()
    {
        return "This is the client-facing website.";
    });
});

Route::group([
    "domain" => "dev.admin.tutorial-laravel-4-multisites"
], function()
{
    Route::any("/about", function()
    {
        return "This is the admin site.";
    });
});
```

> This was extracted from **app/routes.php**.

Aside from the basic routes Laravel 4 supports, it's also possible to group routes within route groups. These groups provide an easy way of applying common logic, filtering and targeting specific domains.

Not only can we explicitly target our different virtual host domains, we can target all subdomains with sub-domain wildcard:

```
Route::group([
    "domain" => "dev.{sub}.tutorial-laravel-4-multisites"
], function()
{
    Route::any("/whoami", function($sub)
    {
        return "You are in the '" . $sub . "' sub-domain.";
    });
});
```

> This was extracted from **app/routes.php**.

This functionality allows some pretty powerful domain-related configuration and logical branching!

Translation

Laravel 4 includes a translation system that can greatly simplify developing multisites. Translated phrases are stored in configuration files, and these can be returned in views.

Using Language Lookups

The simplest example of this requires two steps: we need to add the translated phrases and we need to recall them in a view. To begin with; we're going to add a few English and Dutch phrases to the configuration files:

```
<?php

return [
    "instructions" => "Follow these steps to operate cheese:",
    "step1"        => "Cut the cheese.",
    "step2"        => "Eat the :product!",
    "product"      => "cheese"
];
```

This file should be saved as **app/lang/en/steps.php**.

```
<?php

return [
    "instructions" => "Volg deze stappen om kaas te bedienen:",
    "step1"        => "Snijd de kaas.",
    "step2"        => "Eet de :product!",
    "product"      => "kaas"
];
```

This file should be saved as **app/lang/nl/steps.php**.

```
<?php

class IndexController
extends BaseController
{
    public function indexAction()
    {
        App::setLocale("en");

        if (Input::get("lang") === "nl")
        {
            App::setLocale("nl");
        }

        return View::make("index/index");
    }
}
```

> This file should be saved as **app/controllers/IndexController.php**.

This will toggle the language based on a querystring parameter. The next step is actually using the phrases in views:

```
@extends("layout")
@section("content")
    <h1>
        {{ Lang::get("steps.instructions") }}
    </h1>
    <ol>
        <li>
            {{ trans("steps.step1") }}
        </li>
        <li>
            {{ trans("steps.step2", [
                "product" => trans("steps.product")
            ]) }}
        </li>
    </ol>
@stop
```

> This file should be saved as **app/views/www/layout.blade.php**.

The **Lang::get()** method gets translated phrases out of the configuration files (in this case steps.instructions). The **trans()** method serves as a helpful alias to this.

You may also have noticed that **step2** has a strange **:product** placeholder. This allows the insertion of variable data into translation phrases. We can pass these in the optional second parameter of **Lang::get()** and **trans()**.

> You can learn more about the Localization class at: **http://laravel.com/docs/localization**.

Using Language Lookups in Packages

We're not going to go into the details of how to create packages in Laravel 4, except to say that it's possible to have package-specific translation. If you've set a package up, and registered its service provider in the application configuration, then you should be able to insert the following lines:

```
public function boot()
{
    $this->package("formativ/multisite", "multisite");
}
```

> This was extracted from **workbench/formativ/multisite/src/Formativ/Multisite/MultisiteServiceProvider.php**.

...in the service provider. Your package will probably have a different vendor/package name, and you need to pay particular attention to the second parameter (which is the alias to your package assets).

Add these translation configuration files also:

```
<?php

return [
    "instructions" => "Do these:"
];
```

> This file should be saved as **workbench/formativ/multisite/src/lang/en/steps.php**.

```
<?php

return [
    "instructions" => "Hebben deze:"
];
```

> This file should be saved as **workbench/formativ/multisite/src/lang/nl/steps.php**.

Finally, let's adjust the view to reflect the new translation phrase's location:

```
<h1>
    {{ Lang::get("multisite::steps.instructions") }}
</h1>
```

> This was extracted from **app/views/www/index/index.blade.php**.

It's as easy at that!

> You can learn more about package resources at: **http://laravel.com/docs/packages#package-configuration**.

Caching Language Lookups

Getting translated phrases from the filesystem can be an expensive operation, in a big system. What would be even cooler is if we could use Laravel 4's built-in cache system to make repeated lookups more efficient.

To do this, we need to create a few files, and change some old ones:

```
// 'Lang' => 'Illuminate\Support\Facades\Lang',
"Lang" => "Formativ\Multisite\Facades\Lang",
```

> This was extracted from **app/config/app.php**.

This tells Laravel 4 to load our own Lang facade in place of the one that ships with Laravel 4. We've got to make this facade...

```php
<?php

namespace Formativ\Multisite\Facades;

use Illuminate\Support\Facades\Facade;

class Lang
extends Facade
{
    protected static function getFacadeAccessor()
    {
        return "multisite.translator";
    }
}
```

> This file should be saved as **workbench/formativ/multisite/src/Formativ/Multisite/Facades/Lang.php**.

This is basically the same facade that ships with Laravel 4, but instead of returning **translator** it will return **multisite.translator**. We need to register this in our service provider as well:

```php
public function register()
{
    $this->app["multisite.translator"] =
         $this->app->share(function($app)
    {
        $loader = $app["translation.loader"];
        $locale = $app["config"]["app.locale"];
        $trans  = new Translator($loader, $locale);

        return $trans;
    });
}
```

> This was extracted from **workbench/formativ/multisite/src/Formativ/Multisite/MultisiteServiceProvider.php**.

I've used very similar code to what can be found in **vendor/laravel/framework/src/Illuminate/Translation/TranslationServiceProvider.php**. That's because I'm still using the old file-based loading system to get the translated phrases initially. We'll only return cached data in subsequent retrievals.

Lastly; we need to override how the translator fetches the data.

```
1  <?php
2
3  namespace Formativ\Multisite;
4
5  use Cache;
6  use Illuminate\Translation\Translator as Original;
7
8  class Translator
9  extends Original
10 {
11     public function get($key, array $replace = array(),
12         $locale = null)
13     {
14         $cached = Cache::remember($key, 15,
15             function() use ($key, $replace, $locale)
16         {
17             return parent::get($key, $replace, $local
18         });
19
20         return $cached;
21     }
22 }
```

This file should be saved as **workbench/formativ/multisite/src/Formativ/Multisite/Translator.php**.

The process is as simple as subclassing the Translator class and caching the results of the first call to the **get()** method.

> You can learn more about facades at: **http://laravel.com/docs/facades**.

Creating Multi-Language Routes

Making multi-language routes may seem needless, but link are important to search engines, and the users who have to remember them.

To make multi-language routes, we first need to create some link terms:

```
<?php

return [
    "cheese" => "cheese",
    "create" => "create",
    "update" => "update",
    "delete" => "delete"
];
```

> This file should be saved as **app/lang/en/routes.php**.

```
<?php

return [
    "cheese" => "kaas",
    "create" => "creëren",
    "update" => "bijwerken",
    "delete" => "verwijderen"
];
```

> This file should be saved as **app/lang/nl/routes.php**.

Next, we need to modify the **app/routes.php** file to dynamically create the routes:

```php
$locales = [
    "en",
    "nl"
];

foreach ($locales as $locale)
{
    App::setLocale($locale);

    $cheese = trans("routes.cheese");
    $create = trans("routes.create");
    $update = trans("routes.update");
    $delete = trans("routes.delete");

    Route::any($cheese . "/" . $create,
        function() use ($cheese, $create)
    {
        return $cheese . "/" . $create;
    });

    Route::any($cheese . "/" . $update,
        function() use ($cheese, $update)
    {
        return $cheese . "/" . $update;
    });

    Route::any($cheese . "/" . $delete,
        function() use ($cheese, $delete)
    {
        return $cheese . "/" . $delete;
    });
}
```

> This was extracted from **app/routes.php**.

We hard-code the locale names because it's the most efficient way to return them in the routes file. Routes are determined on each application request, so we dare not do a file lookup…

What this is basically doing is looping through the locales and creating routes based on translated phrases specific to the locale that's been set. It's a simple, but effective, mechanism for implementing

multi-language routes.

If you would like to review the registered routes (for each language), you can run the following command:

```
php artisan routes
```

> You can learn more about routes at: **http://laravel.com/docs/routing**.

Creating Multi-Language Content

Creating multi-language content is nothing more than having a few extra database table fields to hold the language-specific data. To do this; let's make a migration and seeder to populate our database:

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreatePostTable
extends Migration
{
    public function up()
    {
        Schema::create("post", function($table)
        {
            $table->increments("id");
            $table->string("title_en");
            $table->string("title_nl");
            $table->text("content_en");
            $table->text("content_nl");
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists("post");
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_CreatePostTable.php**.

```
<?php

class DatabaseSeeder
extends Seeder
{
    public function run()
    {
        Eloquent::unguard();
        $this->call("PostTableSeeder");
    }
}
```

This file should be saved as **app/database/seeds/DatabaseSeeder.php**.

```
<?php

class PostTableSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $posts = [
            [
                "title_en"   => "Cheese is the best",
                "title_nl"   => "Kaas is de beste",
                "content_en" => "Research has shown...",
                "content_nl" => "Onderzoek heeft aangetoond..."
            ]
        ];
        DB::table("post")->insert($posts);
    }
}
```

> This file should be saved as **app/database/seeds/PostTableSeeder.php**.

To get all of this in the database, we need to check the settings in app/config/database.php and run the following command:

```
php artisan migrate --seed --env=local
```

This should create the post table and insert a single row into it. To access this table/row, we'll make a model:

```
<?php

class Post
extends Eloquent
{
    protected $table = "post";
}
```

> This file should be saved as **app/models/Post.php**.

We'll not make a full set of views, but let's look at what this data looks like straight out of the database. Update your **IndexController** to fetch the first post:

```
<?php

class IndexController
extends BaseController
{
    public function indexAction($sub)
    {
        App::setLocale($sub);
        return View::make("index/index", [
            "post" => Post::first()
        ]);
    }
}
```

> This file should be saved as **app/controllers/IndexController.php**.

Next, update the **index/index.blade.php** template:

```
<h1>
    {{ $post->title_en }}
</h1>
<p>
    {{ $post->content_en }}
</p>
```

> This was extracted from **app/views/www/index/index.blade.php**.

If you managed to successfully run the migrations, have confirmed you have at least one table/row in the database, and made these changes; then you should be seeing the english post content in your **index/index** view.

That's fine for the English site, but what about the other languages? We don't want to have to add additional logic to determine which fields to show. We can't use the translation layer for this either, because the translated phrases are in the database.

The answer is to modify the **Post** model:

```
public function getTitleAttribute()
{
    $locale = App::getLocale();
    $column = "title_" . $locale;
    return $this->{$column};
}

public function getContentAttribute()
{
    $locale = App::getLocale();
    $column = "content_" . $locale;
    return $this->{$column};
}
```

> This was extracted from **app/models/Post.php**.

We've seen these attribute accessors before. They allow us to intercept calls to **$post->title** and **$post->content**, and provide our own return values. In this case; we return the locale-specific field value. Naturally we can adjust the view use:

```
<h1>
    {{ $post->title }}
</h1>
<p>
    {{ $post->content }}
</p>
```

> This was extracted from **app/views/www/index/index.blade.php**.

We can use this in all the domain-specific views, to render locale-specific database data.